Vu Hoang Tuan & Tran Gia Hoang Long

"CUTE" CLIENT FOR HESSIAN

Thesis
CENTRAL OSTROBOTHNIA OF APPLIED SCIENCES
Degree Programme in Information Technology
April 2011

PREFACE

We would like to thank all the people who have guided and inspired us during our Bachelor of Information Technology studies. The Department of Information Technology and Business, the Central Ostrobothnia University of Applied Sciences has been our special home where our knowledge and ideas were obtained and improved. There are many people to acknowledge for their support in our four-year studies.

We would like to express our appreciation to Grzegory Sczewcyk, our thesis advisor for his encouragement and guidance he had given to us throughout this thesis process. We found his supervisions were especially invaluable and helpful for our constantly improving implementation.

We would like to thank Marko Forsell. His advices were invaluable when we were struggling with our thesis's boundary. We would like to thank Esko Johnson for his kind support for the completion of our thesis.

Finally, we would like to thank Steve Montel and Theresa Nguyen, the Chief-Executive-Officer and Director of Sales and Marketing of Caucho Technology for their special encouragement and support.

ABBREVIATIONS & ACRONYMS

RMI	Remote Method Invocation
SUID	Serial version Unique Identifier
RPC	Remote Procedure Call
XML	Extensible Markup Language
IoC	Inversion of Control
AOP	Aspect-Oriented Programming
MVC	Model View Controller
HTTP	Hyper-Text Transfer Protocol
SOAP	Simple Object Access Protocol
WSDL	Web Service Description Language
UDDI	Universal Description, Discovery and Integration
API	Application Program Interface
URL	Uniform Resource Locator
UI	User Interface
OS	Operating System
OSI	Open System Interconnection
JMS	Java Message Service
REST	Representational State Transfer
DCOM	Distributed Component Object Model

Thesis Abstract

Department	**Date**	**Authors**
Technology and Business, Kokkola	20 April 2011	Vu Hoang Tuan & Tran Gia Hoang Long
Degree programme Degree programme in Information Technology		
Name of thesis "Cute" Client for Hessian		
Instructor Grzegorz Szewczyk	**Pages** 53 + Appendices (25)	
Supervisor Grzegorz Szewczyk		

The purpose of this thesis is to present the design and implementation of Hessian binary web service protocol in Qt - a cross platform application and framework. The implementation of Hessian protocol, which can be found in Java, C#, Python, Objective-C and many other platforms, encourages the use of Hessian as a complementary data exchange method of XML-based protocols.

The implementation is built on top of the Qt Network infrastructure that simplifies the management of sending network requests and receiving replies. The module together with other classes and libraries has made the development of the protocol on Qt framework an easy and pleasant experience.

As a partial implementation of an open-source project, many additional features are prepared to improve the performance. The later project is an implementation to be suitable for production with encryption—decryption algorithm, compression together with caching capabilities.

Keywords:

Hessian, web service, binary protocol, Qt, binary web service protocol.

Contents

1 INTRODUCTION

It is true that XML (Extensible Markup Language) is the thing making Web service possible. When a web service is built, it has to be on one way or another way in XML format. Despite its popularity, XML is discouraged to be used as a data exchanging protocol between systems because of its well-known weaknesses which are the following: the syntax requires high bandwidth, consumes too many resources on the client side for parsing when data in binary form must be converted into XML before it is parsed back into its original form on the receiver's side.

Hessian binary web service protocol (HBWSP) was developed by Caucho (Caucho Technology 2007b) and has been released under open source license. Hessian, as a binary protocol, is compact, efficient and low resource consuming, making the web service usable as simple as calling a local method.

The aim of this thesis is to describe the implementation of the HBWSP RPC in the Qt framework, following the specification of Hessian 2.0 protocol. This implementation will be a foundation for further implementation of the protocol. With the cross-platform power of Qt, Hessian’s support range is significantly extended.

The implementation is built on top of the QtNetwork infrastructure. This module provides the QNetworkAccessManager class which manages the sending network requests and receiving replies. It follows the design of the module by the construction of an HCall class which contains the request parameters (method name, parameters), as well as the reply (result, errors and status of the call).

The thesis begins in Chapter 3 describing terminologies and technologies. The heart of the thesis is Chapter 5 covering the design, the requirements, the implementation and the use of the protocol in Qt. Chapter 5 also includes an overview of the design requirements which was specified by the Hessian 2.0 protocol. These requirements will be further investigated and solved. Next, Chapter 7 shows how the implementation can be used in real application.

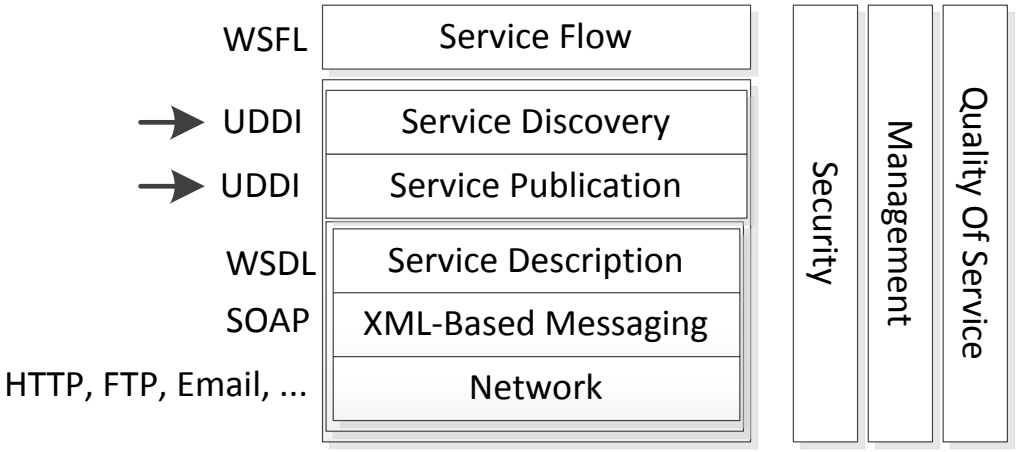
Finally, Chapter 6 presents the result of the work in addition to conclusion and further expansion of the system in Chapter 7.

2 WEB SERVICE BACKGROUND

2.1 Web Services

There are many definitions given for Web Services. They vary from very long one that the World Wide Web Consortium (W3C) defines Web Service: “a software system identified by a URI whose public interfaces and bindings are defined and described using XML” to as simple as “services offered via the Web” (Fisher 2002).

In general, web services are modular software components developed under a set of open standards and are accessible through the Web. The key to success of Web service depends on the intercommunication of the involved systems independently of their natures. In Graph 1, the architecture of web service was described as a stack of multiple layers (Heather 2001):



GRAPH 1. Web Services conceptual stack

The foundation stack is based on the bottom network layer which allows data to be transferred between systems. The messaging layer lies on top of the network layer describing the data formats being transferred. The upper layer, the service description layer, defines the available operations, the valid messages and the access protocol. XML is

used in this stack to leverage the interoperability between systems. Based on the definitions from Web Service Architecture (Heather 2001), here are some common terms:

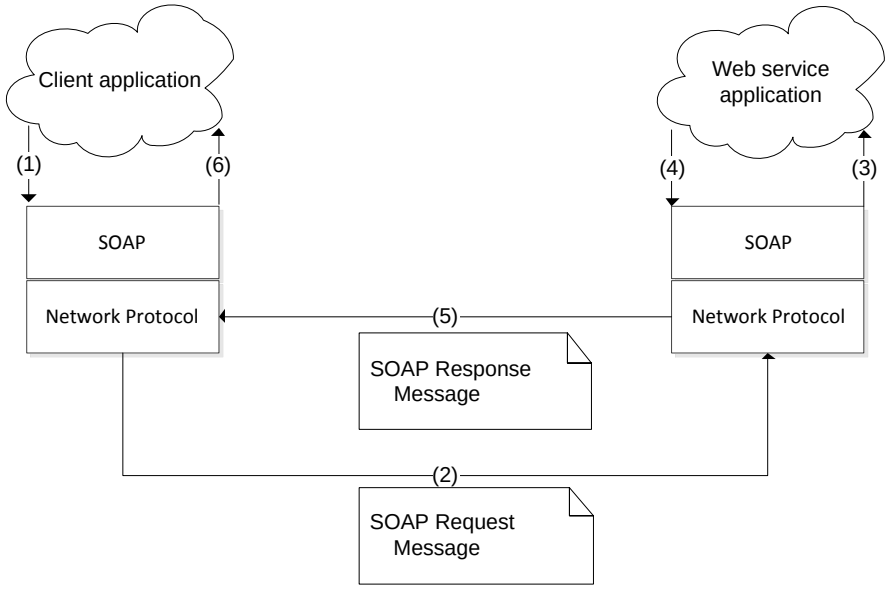
- SOAP – an XML-centric messaging schema that helps sending input and receiving output by exchanging XML documents.
- WSDL – an XML format to describe the service endpoints.
- UDDI – a framework independent of platform to describe public services and interactions with other services.
- WSFL – an XML language for the description of Web Service compositions

2.2 Web service meta-protocols

As the web service evolves, there have been many meta-protocols developed to improve the performance, reliability, maintainability and add to reduce development efforts. They can be characterized by their encoding standards (XML or binary) or communication patterns such as REST, RPC, Messaging, and Streaming (Heather 2001).

2.2.1 XML-based web service

Communications between web service components is usually based on messaging style. As defined in the Web service conceptual stack, SOAP messages – a XML-based structured data - are primarily used to exchange information between Client and Server because of its platform-independence. An example of SOAP message is shown in Graph 2. In a simple case, Client sends a SOAP request to the remote object located on the server, specifying the method and its arguments, the result is returned in a SOAP response message (Heather 2001).



GRAPH 2. SOAP Request Message

According to Web Service Architecture (Heather 2001), application integration with SOAP can be achieved by six steps:

(1) The application creates a request message which contains the Web service operation and optionally arguments as indicated in the service description.
(2) This message, together with the service location is presented to infrastructure where the message is sent out over the network via the underlying network protocol.
(3) The request message is extracted from the SOAP request message and is delivered to the Web service application.
(4) The Web service processes the request message and creates a response.
(5) The SOAP message response then is sent over the network back to the service requestor.
(6) The message is received, processed and delivered to the client application.

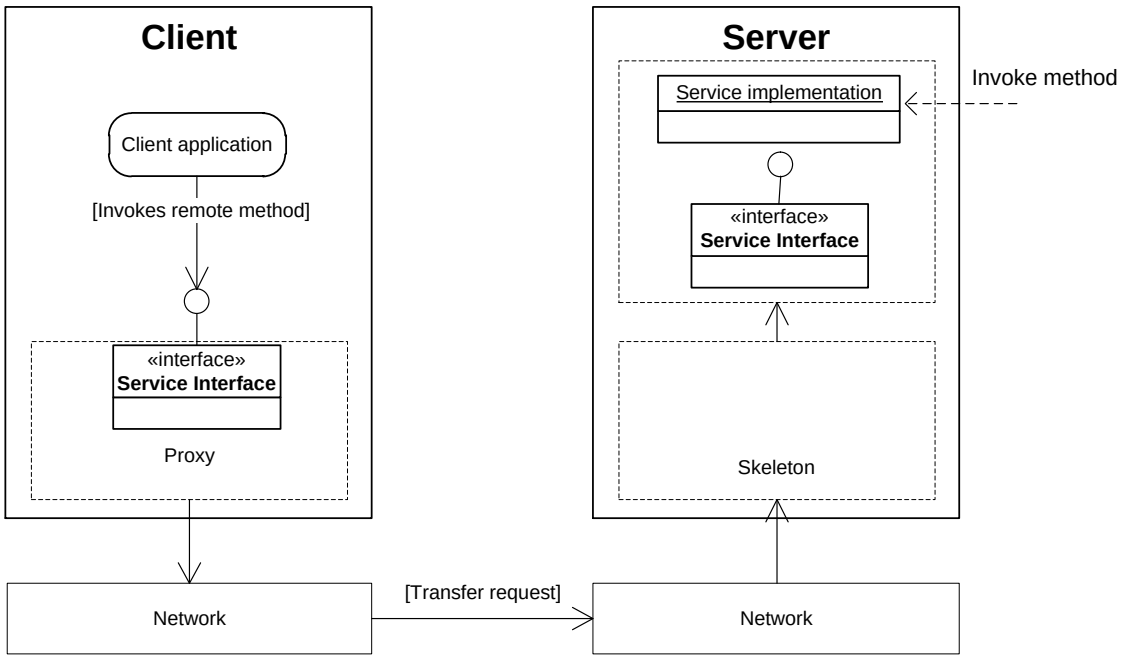
In addition to SOAP, there are a number of commonly used XML-based meta-protocols such as JSON, XML-RPC and Burlap. These protocols suffer from performance degradation when the data need to be parsed and converted into binary form. This process does consume processing power and bandwidth, and scientists have been searching for an effective way to parse XML documents. In 2003, a group from Sun Microsystems

identified the performance problems in current Web Service standards and proposed using binary instead of XML (Paul, Santiago, Kohuske, Marc and Eduardo 2003).

2.2.2 Binary based web service

The need for binary-based encoding protocols arose when there were needs for high performance distributed systems. These systems obviously require high performance and low cost data exchange protocols that XML-based approaches cannot fulfill (Paul et al. 2003). The support for building such complex system varies from complicated industrial standards such as CORBA or Microsoft DCOM, to Java-based solution like RMI (see Appendix 1) or JMS. Such middleware architectures consist of three elements (Sharp 2008):

- The communication element provides a service for message transferring between systems. This layer may involve up to the OSI Transport Layer.
- The middleware element which offers support services to applications
- The application element which contain application logic and user interface.



GRAPH 3. ROI styles of middleware

For example, in Graph 3, the ROI systems, the Proxy containing the Service Interface which is a copied from the Service Interface from the server is set up when binding takes place. This proxy contains all the necessary code for marshalling, un-marshalling as well as security and compression. The corresponding mechanism on the server side is called Skeleton (Sharp 2008).

However, when firewalls are involved, the use of mentioned protocols becomes complicated because they use raw TCP/IP connections in order to transmit data(Ingham, Rees, and Norman 1999). Conventional firewalls block accesses based on host address and port number that make binary communication impossible.

On the other hand, the implementation of those technologies is not widely available. CORBA implementation can be found in C, C++, Java and a wide range of languages but these languages are not intended for the web and sometimes, the implementation have been found to be complex, slow, incompatible and incomplete (Baker 1994).

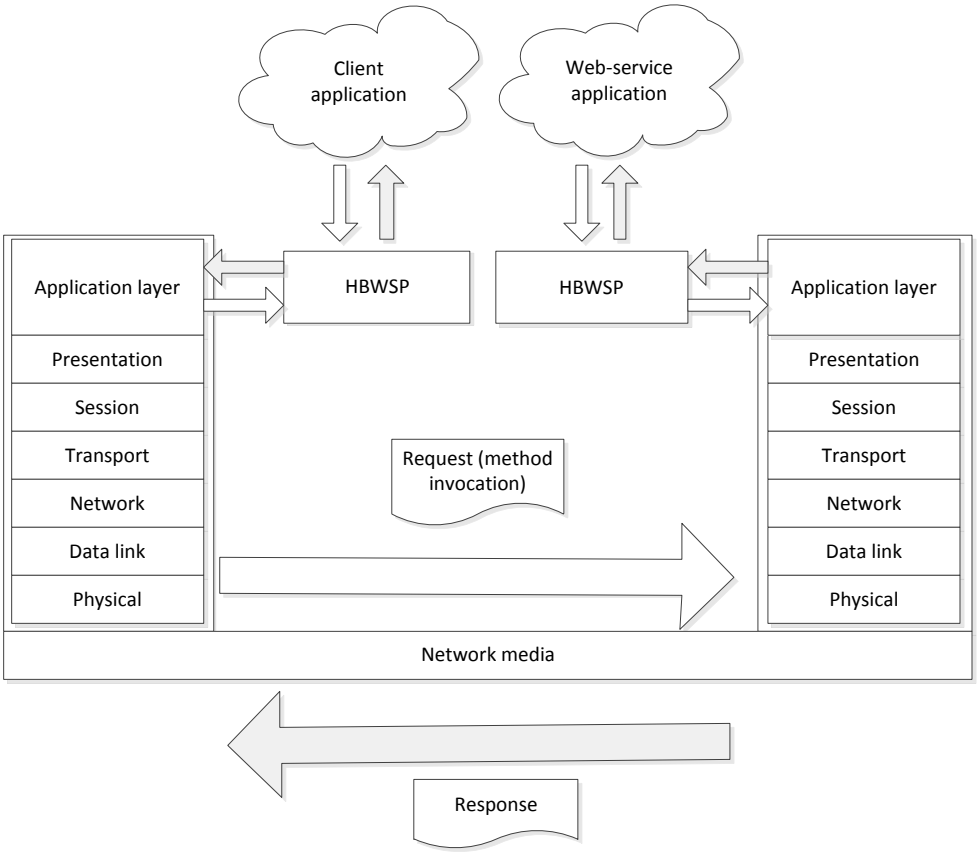
Caucho Technology's approach- the HBWSP- uses binary encoding over HTTP or TCP, Hessian makes the web possible by implementing on a wide range of platforms, including Java, Flash, Python, C++, C#. NET, PHP, Ruby, Objective-C and other languages (Caucho Technology 2007b).

2.3 The Hessian Binary Web Services Protocol

2.3.1 Overview

HBWSP is a cross-platform binary-based RPC intended for web services. HBWSP defines the RPC mechanism, encoding standards and error handling that allow client to execute remote services located on the server. The protocol is mainly based on the HTTP on top

the Application Layer of the OSI model, which is the closest layer to the end user (Caucho Technology 2007b). The HBWSP messaging is shown in Graph 4.



GRAPH 4. HBWSP messaging

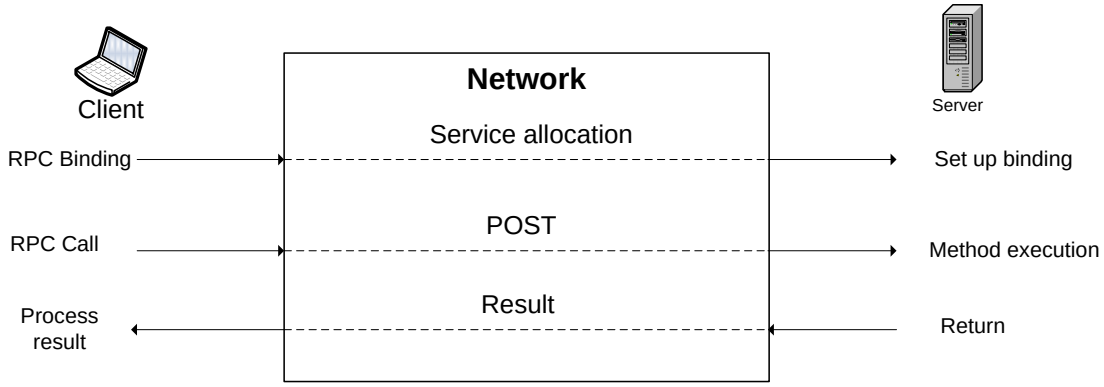
The protocol makes use of a two ways exchange between a Client and a Server. On each exchange, the Client sends a request resource identifying the method to be performed and optional parameters supporting the method. The Server replies the request with a response containing the status of the execution with possible additional information regarding the result (Caucho Technology 2007b).

By using HTTP, the Hessian service is available through the common port 80 that is not blocked by conventional firewall operations. This has not only made Hessian, the only binary protocol that is suitable for web service, but also made the use of Hessian as simple as the XML-based approached. Originally, Hessian supports REST, Messaging and RPC

communication. However, depending on the nature of the application, HBWSP can be implemented on TCP to support Streaming communication patterns (Caucho Technology 2007b). In that case, developers are responsible for the control of connection sessions.

2.3.2 Operations

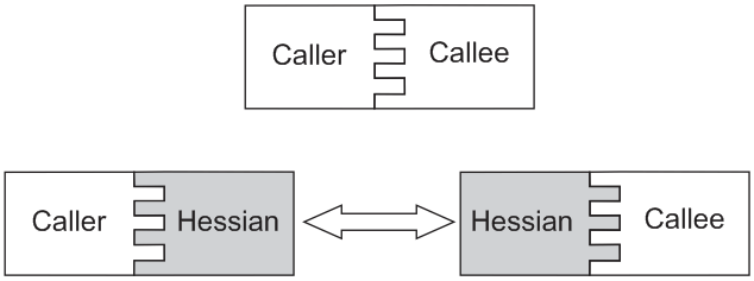
HBWSP is based on a Client-Server model with the traditional asymmetric relationship: the Client can send request to the Server, and Server sends result back, but not vice-versa. In general, the operation of the model can be summarized as follows:



GRAPH 5. HBWSP Client-Server model with RPC

In the beginning, clients must find the servers that offer the service that they are interested in by a process called binding. In that process, clients must be provided the name of the server or the server address or at least an identification to find the appropriate server. Then clients make call to the remote methods via POST requests with data containing method name and supporting arguments (Caucho Technology 2007b).

In a general context, participants are in different address spaces and are inter-connected via a logical network to allow data transferring. Hessian makes the calling of remote methods as simple and executing local procedures (Caucho Technology 2007b).

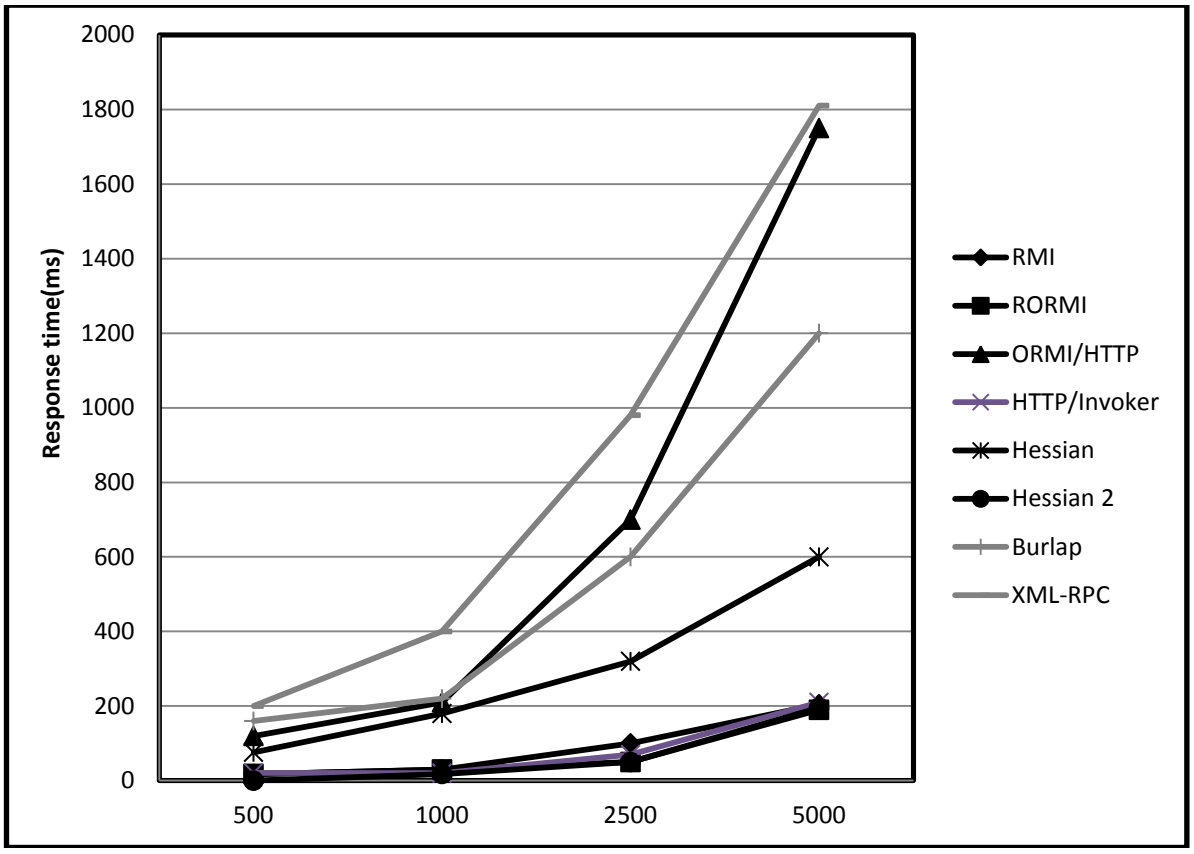


GRAPH 6. Local call (above) and Remote call (below)

In this scenario in Graph 6, both the caller and callee see the remote method via an interface similarly to the local method. The Hessian layer performs the operation known as serialization and (sometimes referred as marshalling (Van de Velde 2007)) to convert method name and parameters into Hessian binary format that suitable for transferring via the communication system. Correspondingly, the other end performs the de-serialization procedure to convert Hessian binary encoded data into the request message and pass to the callee, i.e. un-marshalling (Van de Velde 2007).

2.3.3 HBWSP performance

Based on binary encoding, Hessian obviously outperforms the XML-based approach. Gredler made a performance test between Hessian protocol, Java's RMI, Oracle's ORMI, Spring's HttpInvoker, and three derivations of Apache XML-RPC on the response time of list of elements of different sizes (Gredler 2008). The tests showed that binary protocols were much faster than XML-based protocols, while Hessian's performance and Oracle's one were taking the lead. Moreover, on a research on Inter-process communication technology of Distributed System, Guo showed the advantages of HBWSP over XML for Net Resources sharing by a Java implementation of a Distributed Resource Sharing system (Wei 2006).



GRAPH 7. Response time for large return lists (Gredler, 2008)

2.3.4 Related implementation

Since its release as an open source project, a variety of Hessian's implementation have been developed. Examples of these are available on the Hessian website (Caucho Technology 2007c). Especially the hessiancpp – C++ implementation of Aldratech (Inc, Caucho Technology 2005) on Sourceforge has been converted to a Qt compatible version on Google Code by Caiiiycuk. However, the implementation is neither fully integrated with the Qt framework (QHessian, 2010), nor follows the Qt's design pattern. Caiiiycuk's implementation made use of the old C++ hessiancpp serialization mechanism while Qt added support for the new Qt Network module. However, Qt provides an intuitive way to perform the serialization task that does not require custom serialization process to be developed (Nokia Qt).

3 AIM OF THE WORK

HBWSP differs from other protocols for its light-weight, cross-platform interoperability and un-matched performance. In comparison with CORBA and DCOM, Hessian is much more web-oriented and lightweight (Walton 2005). Compared to XML-based, Hessian Protocol is much more compact, low bandwidth consumption as well as processing power. In an effort to extend its support, the protocol is implemented in Qt cross-platform framework with the main goal to enable Hessian, which is a fast and efficient way to cooperate with Web service on Qt platform in order to:

- Reduce the work needed to develop Web service integrated application in Qt.
- Make Hessian a more multiplatform standard.
- Provide the serialization and de-serialization of primitive data types and an open framework that can be easily extended for further development (Caucho Technology 2007a).

In comparison with the previous implementation of Caiiiycuk, better support is provided to Qt. By applying the Object-Oriented approach, a network manager class is implemented to represent the service and a call class to manage the remote method accessing. Developers will find using remote methods is as easy as accessing local one. On the other hand, asynchronous and synchronous accessing methods are also supported to fit the controversy uses. A simple way is provided to extend the work and develop their custom serialization for complex object types (Nokia Qt).

Although Hessian service provider was written in Java and supports Java-based platforms such as RIFE, Apache Crayen and Danimica, the Spring framework is selected to develop server side test script because Spring framework is emerging as the most popular and powerful one. It contains many features including: the core package providing IoC and DI features. The context package adds support for the internationalization, and the DAO package provides a JDBC-abstraction layer. The ORM package supplies integration for popular ORM APIs, the AOP package, the web package with basic web-oriented integration and the MVC package (Springsource).

More important, Spring supports the development of remote-enable services with RMI, Spring's HTTP invoker, Hessian and Burlap, JAX-RPC and JMS. It is written that when

Spring is used for the context of RMI over HTTP, Hessian is the best option unless you are dealing with complex object models (Harrop and Machacek 2005).

By implementing the protocol in Qt, a cross-platform and UI framework, developers are able to code Web-service-enabled application and deploy across desktop, mobile and embedded OS without rewriting source code. Qt supports a variable range of platforms such as Embedded Linux, Mac OS X, Windows, Linux/X11, Windows CE/Mobile, Symbian and MeeGo (Nokia Qt).

4 TECHNICAL DOCUMENTATION OF THE IMPLEMENTATION

4.1 System requirement capture

4.1.1 System user need

Goal

The project aims at an implementation of Hessian Binary Web Service Protocol that would ease the development of Web-service integrated applications in Qt. The implementation gives basic supports for primitive data types and provides the foundation for complex object types. The implementation also must comply with the new Qt network module.

Environment

The implementation is not intended to be deployed into business application as it lacks of security, encryption mechanism and the serialization of complex object types. The implementation is intended for academic and testing purposes, and is opened for further extensions to support industrial standard application.

The implementation will be able to complied and executed on most Qt-supported operating systems including Windows, Linux, Mac OS, and mobile OS such as Symbian or Maemo. The client system must also be equipped with a suitable network connection, preferably LAN, not WLAN.

4.1.2 Requirement specification

Performance needs

The implementation aims at equal or slightly surpasses Java's implementation in the performance test.

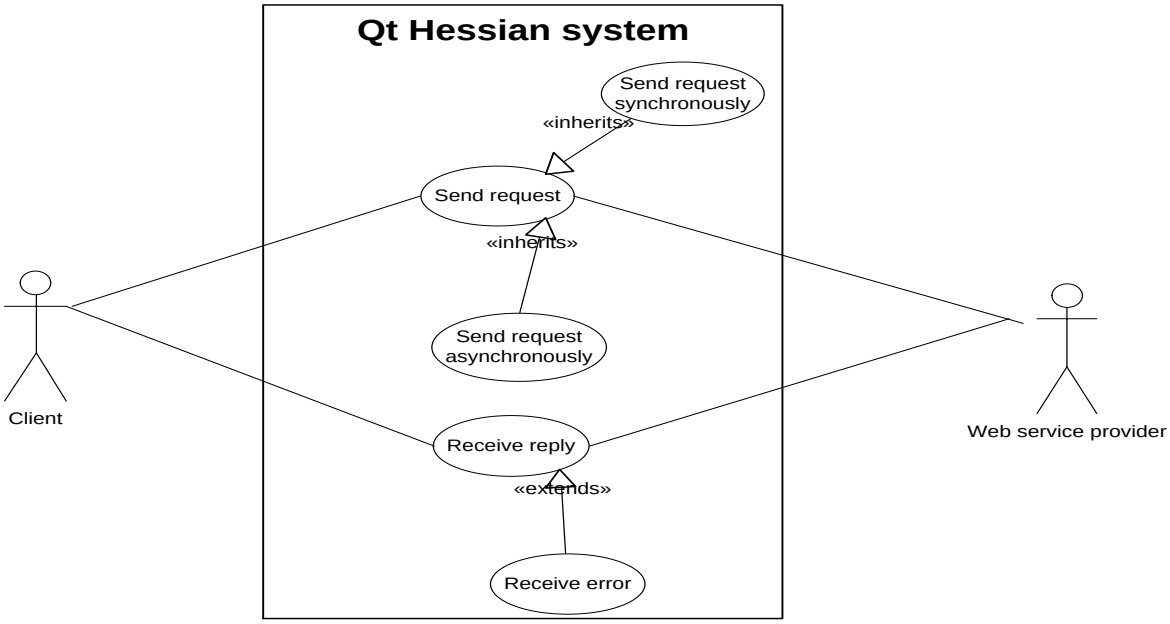
User stories: making synchronous request

Developers want to make a synchronous request. An instance of the system is created with the operation mode SYNCHRONOUS, the URL specifying the service location, the service method and supporting arguments. When a request is sent to the system, it will wait until the result is returned.

User stories: making asynchronous request

Developers want to make an asynchronous request. An instance of the system is created with the operation mode ASYNCHRONOUS, the URL specifying the service location, the service method and supporting arguments. When a request is sent to the system, it continues the routine. When the result is returned, a block of code is executed.

System use cases



GRAPH 8. Use Case Diagram

The use cases diagram in Graph 8 describes the system use cases from the perspective of Client application that is using the system to implement the Web service integration. The following tables describe the system's operation. Table 1 shows synchronous request. Table 2 demonstrate asynchronous request. Table 3 and Table 4 demonstrate the process of receiving reply and receiving error correspondingly.

TABLE 1. Sending synchronous request

Use Case Name	Send request synchronously
Code	1
Version	1.0
Summary	Describe the sending request procedure
Frequency	On every remote method access
Actors	Client application and Web Service provider
Pre-condition	The system is in synchronous mode. The call is in IDLE state
Description	Firstly, user creates the necessary objects and constructs the call. Then, system will send the request to the Service provider

	[ServerException]. Next, system receives the reply from Service provider [InvalidFormatException]. Finally, client application receives the result [See Use case 3].
Exception Path	ServerException: server has thrown exception. System saves the message and change status of the call to ERROR. InvalidFormatException: Received reply data is in bad format. System set status of the call to error.
Post-condition	The call is in FINISHED state with possible result stored.

TABLE 2. Sending asynchronous request

Use Case Name	Send request asynchronously
Code	2
Version	1.0
Summary	Describe the sending procedure in asynchronous mode
Frequency	On every remote method access
Actors	Client application and Web Service provider
Pre-condition	The system is in synchronous mode. The call is in IDLE state
Description	Initially, user needs to create necessary objects and construct the call with appropriate call back. Then, the system will send the request to the Service provider [Server Exception] and receive the reply from Service provider [Invalid FormatException]. Next, client application receives the result via the call-back. [See Use case 3].
Exception Path	ServerException: server has thrown exception. System saves the message and change status of the call to ERROR. InvalidFormatException: Received reply data is in bad format. System set status of the call to error.
Post-condition	The call is in FINISHED state with possible result stored.

TABLE 3. Receiving reply

Use Case Name	Receive reply
Code	3
Version	1.0
Summary	Describe the procedure to extract the result
Frequency	According to needs
Actors	Client application and Web Service provider
Pre-condition	The call is in FINISHED state with result.
Description	Client application access the call for the result, then the result is returned in the wrapper.
Exception Path	
Post-condition	

TABLE 4. Receiving error

Use Case Name	Receive error
Code	4
Version	1.0
Summary	Describe the procedure to get the error
Frequency	According to needs
Actors	Client application and Web Service provider
Pre-condition	The call is in ERROR state with error message.
Description	Client application access the call for the error message, and the result will be returned in the wrapper.
Exception Path	
Post-condition	

Non-functional requirements

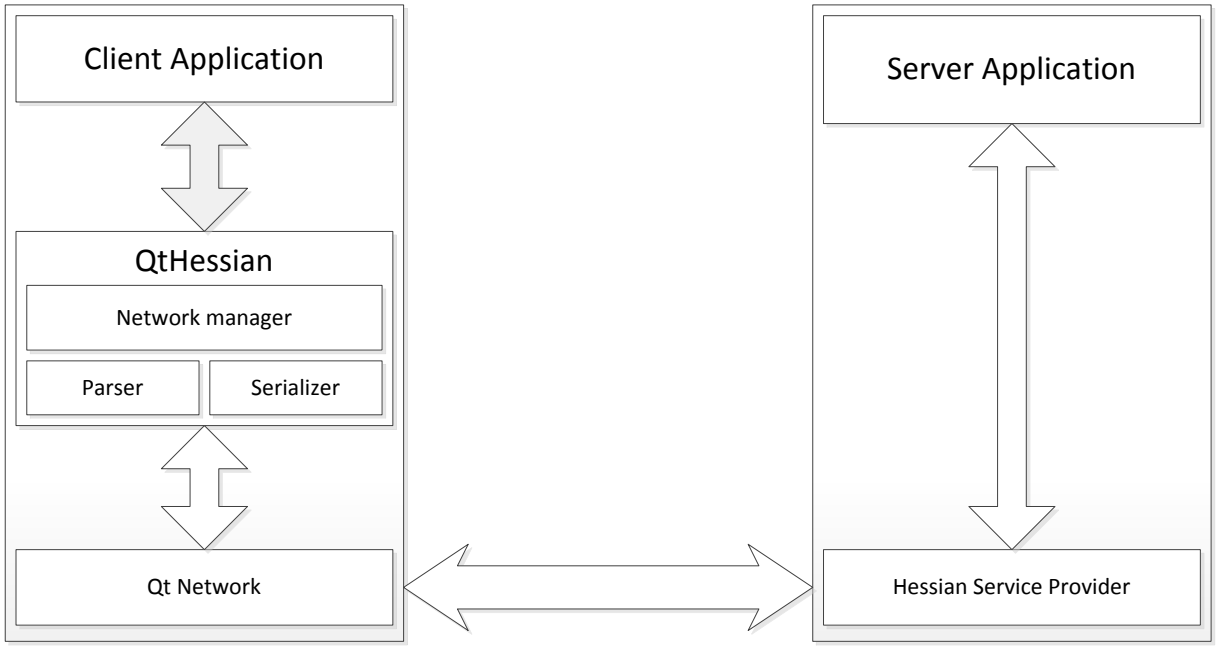
Firstly, for usability, the implementation should make the use of remote service as simple as calling local methods. Secondly, to have good performance, it must match or surpass corresponding Java implementation. Lastly, for expendability, it must be open for further extension.

Environmental requirements

For hardware, the system must be able to run QtCreator and Spring Source Tool Suite. In particular, system speed must exceed 1Ghz and have at least 1GB RAM. Recommended system should be 2GHz CPU and more than 2GB RAM. Next, the implementation requires QtSDK version 4.7, Java SDK 1.6 and Apache Tomcat 6 in order to execute the test script with Spring.

4.2 Requirement analysis and system design

4.2.1 Design overview



GRAPH 9. System structure and communications

The implementation shown in Graph 9 is centralized around three components, the parser unit, the serialization unit and the network unit.

The parser unit and the serialization unit follow the Hessian 2.0 Serialization and Web Services Protocol. The serialization unit writes data to a buffer (QBuffer object) using QDataStream mechanism, which then is included into the post request. The parser unit, on the other hand, parses the QNetworkReply sequentially as a QIODevice.

The network unit uses the structural pattern of Qt Network API, sending network request with QNetworkRequest and receiving response as an instance of QNetworkReply class. From the user point of view, they will only see the system as a service provider and the method to be called. The current implementation does not support streaming mechanism and the data is parsed and returned only when signal *finshed()* is emitted.

The process of communication starts when the Qt application contacts with our API, the API serializes the request with HOutputStream and sends it to the Hessian API on the Server over HTTP through binary communication. Hessian Service Provider parses the request and executes appropriate procedure before sending the result back. When the response reaches the client, it will be parsed with HInputStream. The binary communication has been proved to be better than XML because the serialization and de-

serialization process consume less time while processing power as well as bandwidth. On the other hand, it supports common communication protocols such as Messaging, RPC, and Streaming, allowing a flexible and extended use. Moreover, this binary translation can be compressed and encrypted, thus saving resource used for transmission and enhancing the application security.

4.2.2 Class selection process

The description for *Send request synchronously* use case in Table 1 is: the user creates the necessary objects and constructs the call. Then, the system will send the request to the Service provider. Next, it receives the reply from Service Provider. Finally, the client application receives the result.

Second description is for *Send request asynchronously* use case in Table 2: Initially, the user needs to create necessary objects and construct the call with appropriate call back. Then, the system will send the request to the Service provider and receive the reply from Service provider. Next, the client application receives the result via the call-back.

Next, the *Receive reply* use case in Table 3 is described as follows: At first, the client application accesses the call for the result, then the result is returned in the wrapper.

Finally, we describe the *Receive error* in Table 4: the Client application accesses the call for the error message, and the result will be returned in the wrapper.

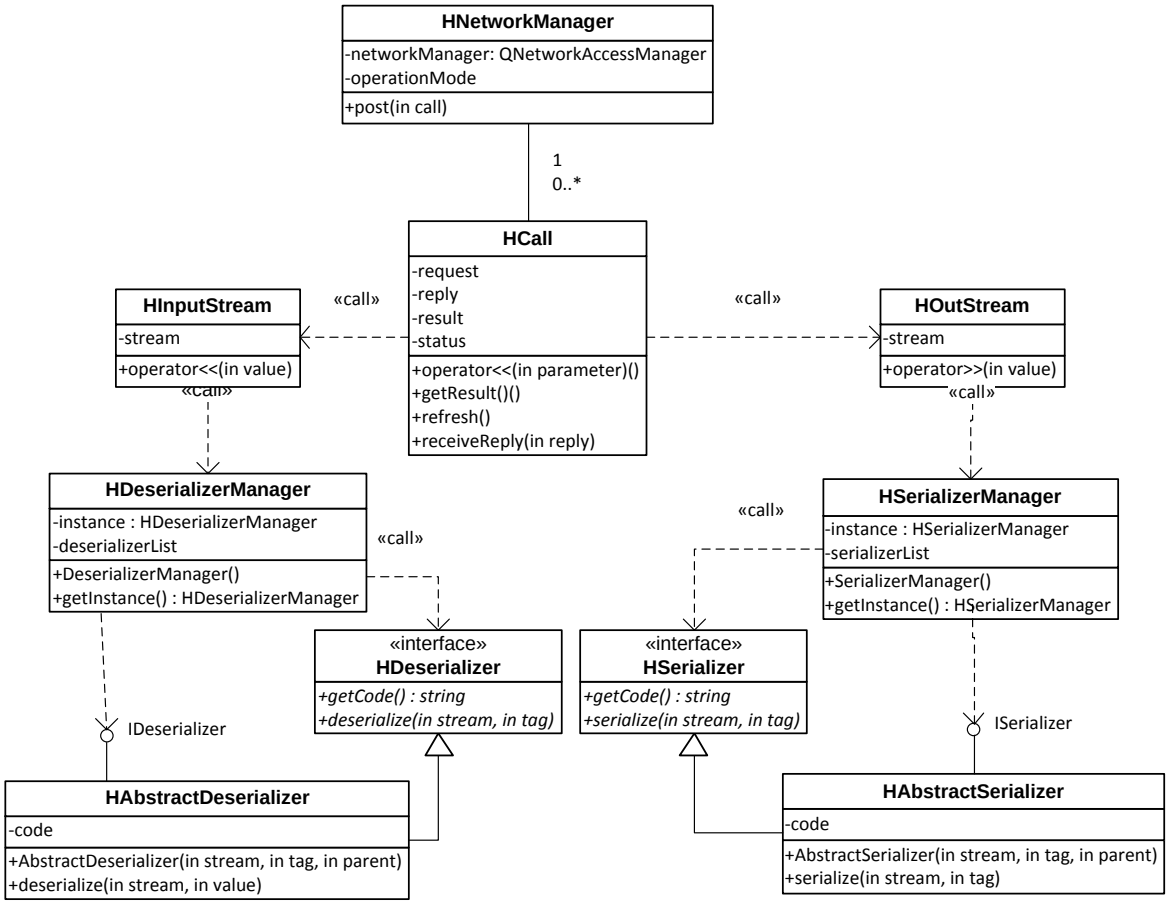
Observed nouns:

user	objects	call	system	request	Service provider
reply	result	callback	wrapper	error	Client application

Preliminary analysis yields the following list of classes and attributes. For classes, there are Call and Wrapper class. The attributes are made up of request, reply, result, error message and call back.

4.2.3 Design class structure

System structure is described in the following class diagram in Graph 10:



GRAPH 10. Class diagram

Class description

The classes are made up of HAbstractDeserializer, HAbstractSerializer, HCall, HDeserializer, HDeserializerManager, HInputStream, HOutputStream, HNetworkManager, HSerializer and HSerializerManager. We will describe them one by one.

First, the HAbstractDeserializer is a custom abstract deserializer class required to derive from this class. Its constructor will register the parser object with DeserializerManager.

Second, the HAbstractSerializer is a custom serializer class required to derive from this class. Its constructor will register the serializer object with SerializerManager.

Third, HCall class acts as an interface to the system and interacts directly with the user. Through this class, the user will construct parameters of a call, make the requests and receive the replies as well as errors. Some of the synonyms can be Call, Remote method. In addition, Call object can be reused by modifying the parameters and call method *refresh()*.

Fourth, the HDeserializer class manages custom deserializer objects created by users. It can be called deserializer interface. Moreover, it implements Singleton for global access and initialization on request.

Next, the HDeserializerManager class manages custom deserializer objects created by users. It is also known as custom parser manager. It implements Singleton for global access and initialization on request.

Next, when the data in Hessian binary form is sent back from the server, it was de-serialized into Qt binary form by HInput class, which is also known as Parser or Deserializer. It also calls the DeserializerManager when the users implement their own parsers.

Next, HNetworkManager class, or network manager or service endpoint is described. It represents the service endpoint, each web service is identified by an URL. It is dependent on QNetworkAccessManger class from Qt network package for networking operations.

The eighth class is HOutputStream or Serializer. When the user invokes a call, its data is converted from Qt binary form to Hessian binary format before sending. It also calls the SerializerManager when users implement their own serializer.

The next class is the HSerializer or Serializer interface. This class manages custom serializer objects created by users. It also implements Singleton for global access and initialization on request.

The last class is HSerializerManager or Serializer. This class manages custom serializer objects created by users. It also implements Singleton for global access and initialization on request.

Class attributes

The following table describes the attributes of above classes. It also shows the forms which we are using in the implementation.

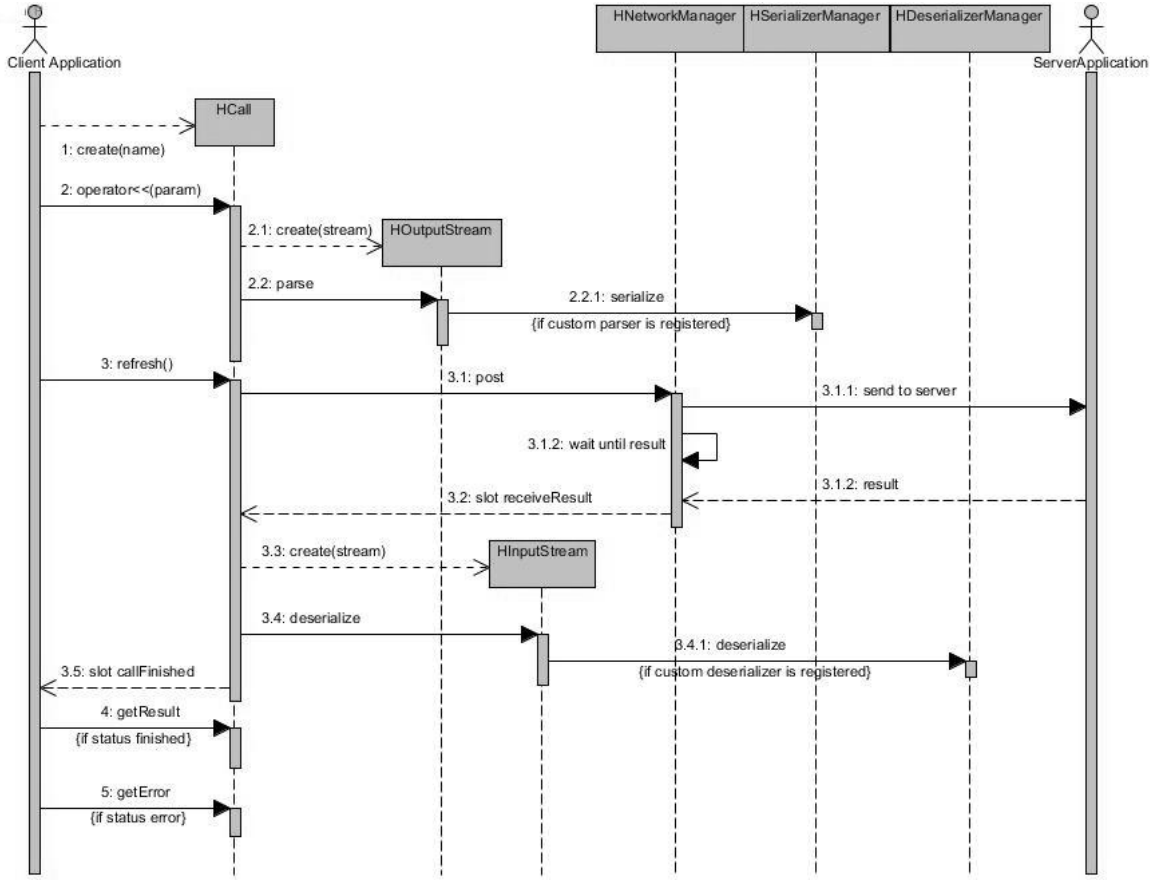
TABLE 5. Class attributes and description

Class/Attribute		Description	Form
HAbstractDeserializer			
	code	The signature of the deserializer, identifies which deserializer to be used by the HDeserializerManager	QString
HAbstractSerializer			
	code	The signature of the serializer, identifies which serializer to be used by the HSerializerManager.	QString
HCall			
	request	The request object containing information of the call to send to the Service Provider by the QNetworkAccessManager.	QNetworkRequest
	reply	The reply object containing the information returned from the call.	QNetworkReply
	result	The result returned from the call wrapped with QVariant	QVariant*
	status	Status of the call	enum
HDeserializerManager			
	instance	The static instance of HDeserializerManager	HDeserializerManager
	deserializerList	The map of deserializer objects	QMap
HInputStream			
	stream	The data stream contains the returning binary	QDataStream

HNetworkManager			
	networkManager	The network manager which based on Qt Network package	QNetworkAccessMa-nager
	operationMode	The operation mode of the network manager, which is either synchronous or asynchronous	enum
HOutputStream			
	stream	The datastream which is going to be sent	QDataStream
HSerializerManager			
	instance	The static instance of the HSerializerManager	HSerializerManager
	serializerList	The map of serializer objects	QMap

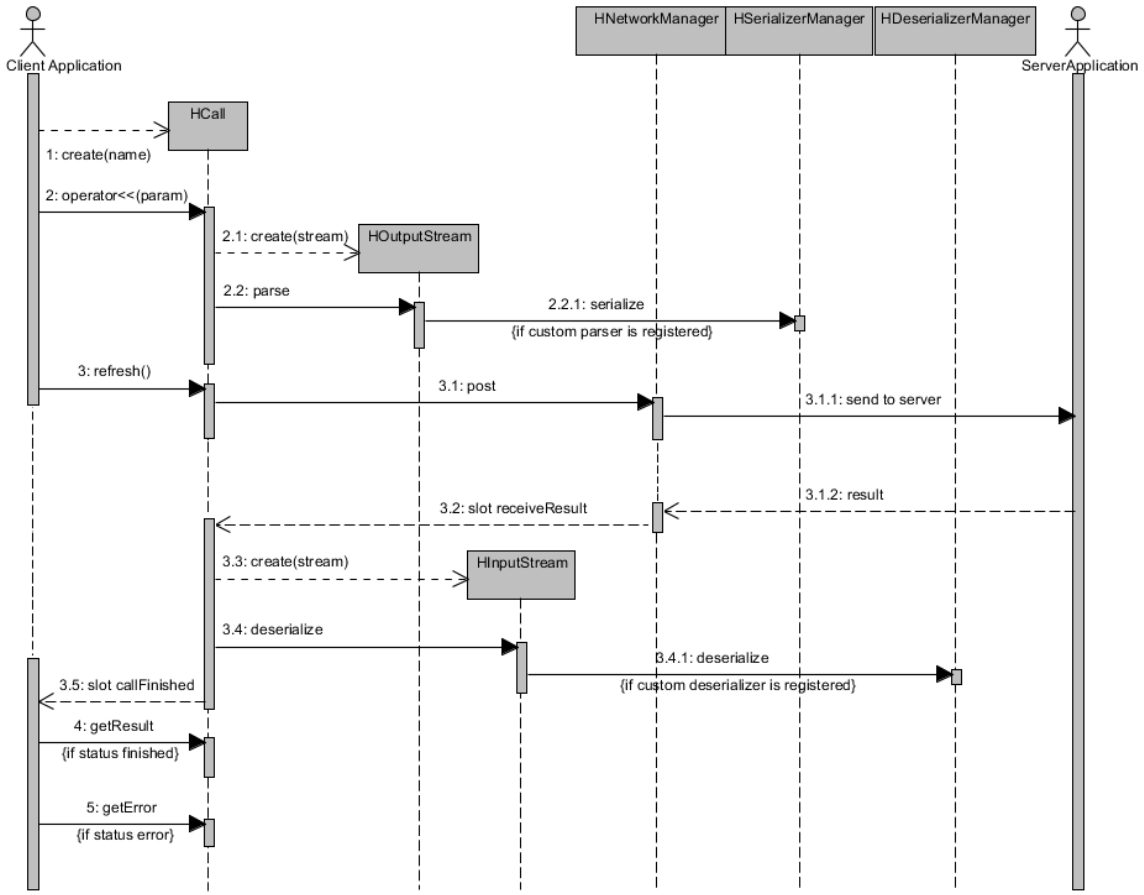
4.2.4 Behavioral design

Sequence diagrams



GRAPH 11. Operation mode SYNCHRONOUS for calling

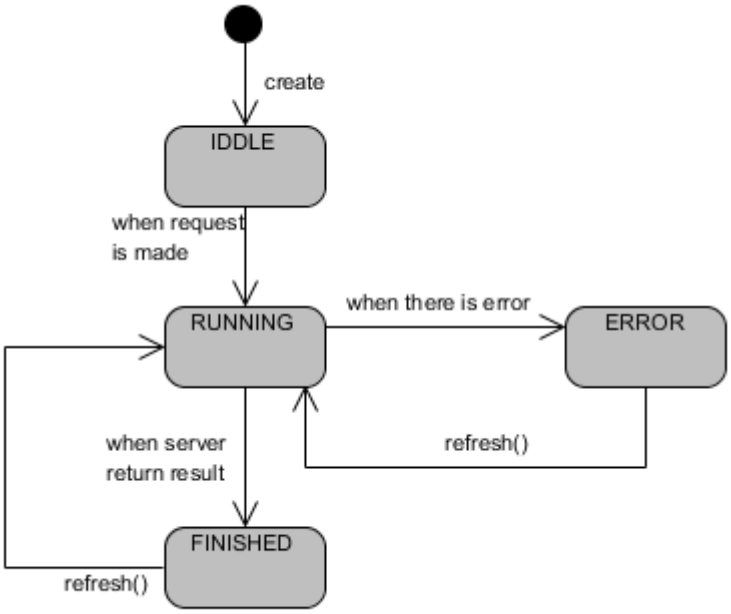
This sequence diagram in Graph 11 shows how the system calls in operation mode synchronous. As we can see, the Client Application made a request to the Server. The request is processed by HCall then sent over HNetworkManager to the Server Application. In this mode, the HNetoworkManger will wait until result before sending to the Server Application. After processing, the Server will send out the result back to the Client.



GRAPH 12. Operation mode ASYNCHRONOUS for calling

This sequence diagram in Graph 12 shows how the system calls in operation mode asynchronous. As we can see, the Client Application made a request to the Server. The request is processed by HCall then sent over HNetworkManager to the Server Application. After processing, the Server will send out the result back to the Client.

Statecharts



GRAPH 13. Statechart for the class HCall

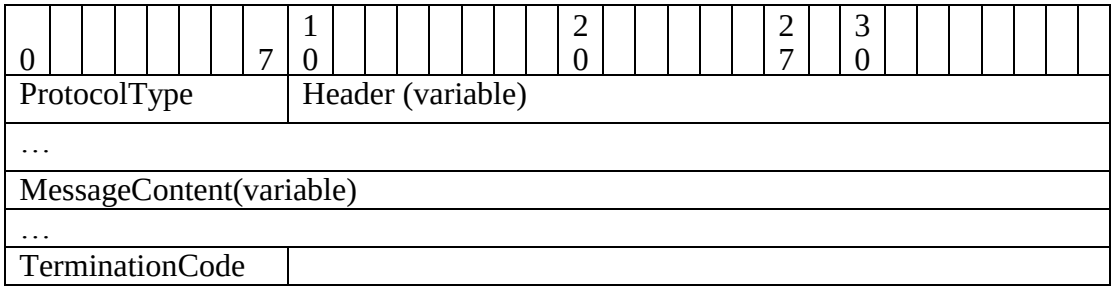
The Statechart in Graph 13 shows how HCall works in our system. First, it's in Iddle state, when a request is made, it begins to process in Running state. If there is any error, it comes to Error state. When server return result, it will be in Finished state.

4.3 Implementation

4.3.1 Hessian message construction

In this part, the message frame is shown. Following it, a table is constructed to describe more about the frame.

General message frame structure:



GRAPH 9. General message frame

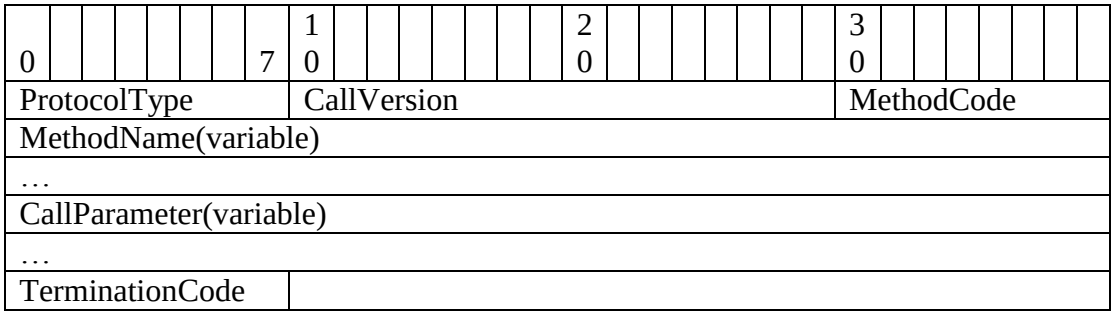
The frame in Graph 14 contains the ProtocolType, the Header, the MessageContent and the TerminationCode. The ProtocolType has the size of 1 byte containing a character that identifies the protocol type. The size of Header is variable and contains support information of the ProtocolType regarding MessageContent which contains stream of bytes that represent content. The size of MessageContent is also variable. The last one, TerminationCode, has 1 byte in size signatures the end of message. It has the value of 'z' or 'Z'. Table 6 shows a summary of those.

TABLE 6. Message frame description

Name	Size	Description	Value
ProtocolType	1 byte	This field contains a character that identifies the protocol type	
Header	variable	This field follows the ProtocolType contains support information of the ProtocolType regarding the MessageContent	
MessageContent	variable	This field contains stream of bytes that represent the content. The structure of the bytes must be defined in the Header	
TerminationCode	1 byte	This 1 byte field signatures the end of the message.	‘z’ or ‘Z’

Hessian RPC 2.0 Call(compliant to Hessian 1.0)

This type of message is sent by default from any Hessian client. It is intended to make the Hessian 2.0 client to be compatible with the Hessian 1.0 Server.



GRAPH 10. Hessian message call

The frame in Graph 15 contains the ProtocolType (1 byte size), the CallVersion (2 bytes size), the MethodCode (1 byte size), the MethodName (variable size), the CallParameter (variable size) and the TerminationCode (1 byte size). The ProtocolType which has the value 'c' containing the character identifying the Call type while the CallVersion (value x0200) identifies the call version. The MethodCode (value 'm') indicates the next field is MethodName representing the remote method name as serialized string following the Hessian format. The CallParameter contains the parameters that support the call. The TerminationCode signatures the end of the message. Table 7 shows a summary of those.

TABLE 7. Message call description

Name	Size	Description	Value
ProtocolType	1 byte	This field contains the character identifying the Call type.	‘c’
CallVersion	2 bytes	This field identifies the version of the call.	x0200
MethodCode	1 byte	It indicates that the following field is the method name.	‘m’
MethodName	variable	This field represents the remote method name as a serialized string following the format defined by Hessian 1.0 Protocol.	
CallParameter	variable	This field contains the parameters that support the call. The variable must be serialized using in the exact order that is defined in the remote interface: from left most variable must be serialized first. This filed may be empty if the remote method does not require any arguments. The variable must be serialized using the Hessian 1.0 Protocol in order to be compatible	

		with Hessian 1.0 Service Provider.	
TerminationCode	1 byte	This 1 byte field signatures the end of the message.	‘z’

Hessian RPC 2.0 Call

0							7	10							20							30						
ProtocolType								CallVersion														OperationType						
MethodName(variable)																												
…																												
ArgumentAcount(count)																												
…																												
CallParameter(variable)																												
…																												

GRAPH 11. Hessian message

The frame in Graph 16 contains the ProtocolType (1 byte size), the CallVersion (2 bytes size), the OperationType (1 byte size), the MethodName (variable size), the ArgumentCount (variable size), the CallParameter (variable size) and the TerminationCode (1 byte size). The ProtocolType contains the character identifying the call type while the CallVersion identifies the call version. The OperationType indicates this message is a RPC. The MethodName represents the method name as serialized string while the ArgumentCount represents an 32-bit integer encoded with Hessian 2.0 protocol. The CallParameter contains the parameters supporting the call, and the TerminationCode signatures the end of the message. Table 8 shows a summary of those.

TABLE 8. Message description

Name	Size	Description	Value
ProtocolType	1 byte	This field contains the character identifying the Call type.	‘H’
CallVersion	2 byte	This field identifies the version of the call.	x0200
OperationType	1 byte	. It indicates that this message is a RPC.	‘C’

MethodName	variable	This field represents the remote method name as a serialized string following the format defined byte Hessian 2.0 Protocol (see Appendix 2/4).	
ArgumentCount	variable	This field represents an 32bit integer that was encoded with Hessian 2.0 Protocol. It tells how many arguments the message carries.	
CallParameter	variable	This field contain the parameters that support the call. The variable must be serialized using Hessian 2.0 Protocol in the exact order that is defined in the remote interface: from left most variable must be serialized first. This filed may be empty if the remote method does not require any arguments.	
TerminationCode	1 byte	This 1 byte field signatures the end of the message.	'z'

Hessian RPC 2.0 Reply

0							7	10							20							30							
ProtocolType								ReplyVersion														ResultCode							
Result(variable)																													
…																													

GRAPH 12. Hessian message reply

The frame in Graph 17 contains the ProtocolType (1 byte size), the ReplyVersion (2 bytes size), the ResultCode (1 byte size) and the Result (variable size). The ProtocolType contains the character identifying the Call type while the ReplyVersion identifies the call version. The ResultCode indicates this message is RPC while the Result contains the result of the call. Table 9 shows a summary of those.

TABLE 9. Message reply description

Name	Size	Description	Value
ProtocolType	1 byte	This field contains the character identifying the Call type.	'H'
ReplyVersion	2 byte	This field identifies the version of the call.	x0200

ResultCode	1 byte	It indicates that this message is a RPC.	‘R’
Result	variable	This field contains the result of the call. The return value is serialized using Hessian 2.0 Protocol and need to be de-serialized before use.	

Hessian RPC 2.0 Fault Reply

0							7	10							20								30							
FailureByteCode								PrototypeType							MessageMap(variable)														
...																													
TerminationCode																													

GRAPH 13. Hessian message fault reply

The frame in Graph 18 contains the FailureByteCode (1 byte size), the ProtocolType (1 byte size), the MessageMap (variable size) and the TerminationCode (1 byte size). The FailureByteCode contains the character indicating the method call was not successful. The ProtocolType identifies the Hessian message while the MessageMap shows the fault in a map of supportive information such as code, message and detail. The TerminationCode contains the result of the call. Table 10 shows a summary of those.

TABLE 10. Message fault reply description

Name	Size	Description	Value
FailureByteCode	1 byte	This field contains the character indicating that the method call was not successful because of exceptions on the server	‘F’
ProtocolType	1 byte	This field identified the Hessian message	‘H’
MessageMap	variable	This field represents the fault in a map of supportive information such as code, message and detail. The map is encoded with Hessian 2.0 Protocol.	‘R’
TerminationCode	1 byte	This field contains the result of the call. The return value is serialized using Hessian 2.0 Protocol and need to be de-serialized before use.	

4.3.2 Network manager

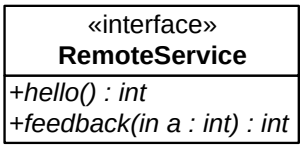
HNetworkManager – the service endpoint

The network manager was built in order to make use of the well-designed QtNetwork package. The network manager represents for the service endpoint and is identified by an URL given in either QString or characters array. The network manager, therefore, requires the service address in its constructors.

The network manager allows making requests in both synchronous mode and asynchronous mode. In synchronous mode, each request is blocked until the result is fully fetched. In asynchronous mode, signal *callFinished(HCall*)* is emitted on each completion and any slots connected that signal will be triggered. The user is responsible for connecting the required slot for processing the received result. The user is freely to access to the QNetworkReply slots to gather information about the request such as download progress, error handling. (Refer Appendix 6 for more information)

HCall – the remote method

If the network manager is the representative of the service endpoint, an object of HCall class represents the remote method that can be accessed from that service. The method must be specified in the Remote Interface in order to be accessed remotely (Refer to Appendix 1 for more information). For example, if the Remote Interface define two remote methods *hello* and *feedback(int)* of the RemoteService:



GRAPH 14. Remote Service

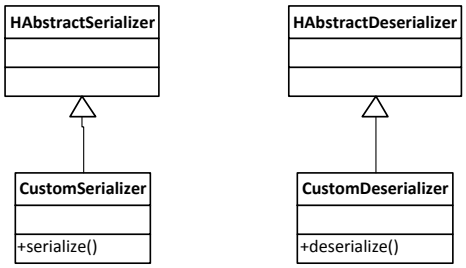
A method with variable number of arguments is discouraged to be used remotely. If a call is finished when the network manager is working in asynchronous mode, it will emit *callFinished()* signal.

4.3.3 Serialization and deserialization

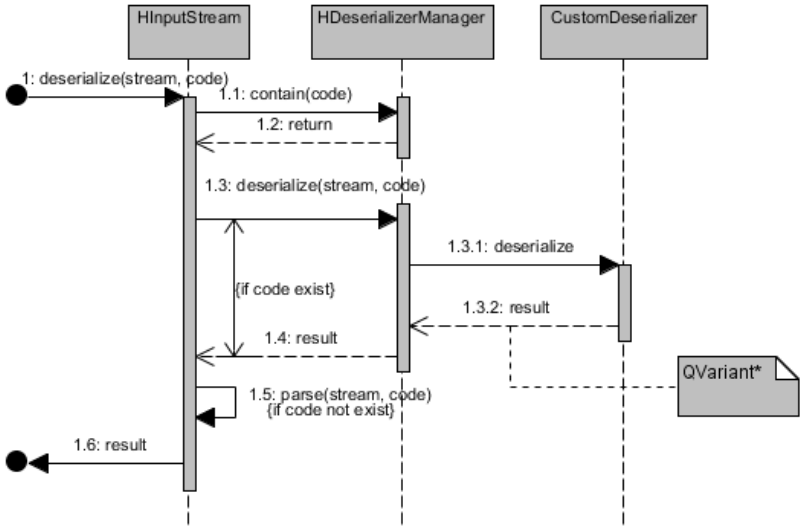
HOutputStream is responsible for serializing data from Qt types to Hessian types following the Hessian 2.0 Protocol (Appendix 2). HInputStream, on the other hand, parses the result from the returned bytes into Qt types. The result is enclosed in a QVariant object and later processed by the user.

The serialization and de-serialization processes are fairly similar. The two classes serializer and deserializer are shown in GRAPH 20. In the beginning, for the serialization procedure, a memory slot is allocated in adequate to the number of blocks required. Then the slot is filled with the serialized data following the Hessian Protocol and then written to the stream at once (Refer to Appendix 2). In order to the implement custom serialization unit, the developer need to create a derived class of the HAbstractSerializer and call the parent constructor to register the class with the built-in singleton serialization unit (see Graph 22).

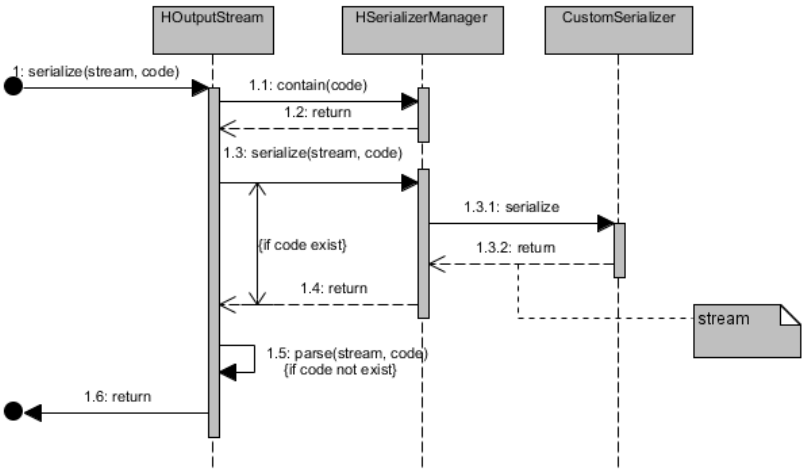
For the de-serialization routine, the class reads the very first byte and then maps that byte to a bytemap to get the proper parsing procedure (see Appendix 2/5-2/8 for bytemap list). The parsed result, finally, is wrapped by a QVariant union object and returned. In order to implement custom serialization units, the developer need to create a derived class of the HAbstractDeserializer and call the parent constructor to register the class with the built-in singleton deserialization unit (see Graph 21).



GRAPH 15. Custom serializer and deserializer class



GRAPH 21. Deserializer Sequence diagram



GRAPH 22. Serializer Sequence diagram

4.4 Project User Guide

4.4.1 System requirements

The implementation is built on Qt 4.7 and Spring framework 3.0.5 with reference to Hessian java client 4.0.7. However, to build a test system, additional software is required such as an IDE (Springsource Tool Suite - STS), Apache Tomcat.

TABLE 11. Software requirements

Software	Version
Qt	4.7+
Spring	3.0+
Hessian client	4.0+
STS	2.6+
Apache Tomcat	6.0+

4.4.2 Project installation

Qt is available for download from http://qt.nokia.com/downloads. Qt is released for free under LGPL license for non-commercial purposes.

The Spring framework is available to download from http://www.springsource.org/download. However, the user is recommended to apply maven to manage Spring's dependencies.

Hessian is also available to download from http://hessian.caucho.com/. User also can use maven to download Hessian jar file.

TABLE 12. Some installation specifications

Compatibility	JDK 1.6
IDE	(STS) SpringSource Tool Suite (http://www.springsource.com/landing/best-development-tool-enterprise-java)
Subversion control	Subversion Tigris (http://subversion.tigris.org/), SmartSVN (http://www.syntevo.com/smartsvn/index.html)
Web server	SpringSource tc server Apache Tomcat

4.4.3 Instruction for the user to get the latest release

The Qt project is available for svn checkout from http://svn3.xp-dev.com/svn/qthessian/.

The Spring test project is available for svn checkout from http://svn3.xp-dev.com/svn/SpringHessianTest/

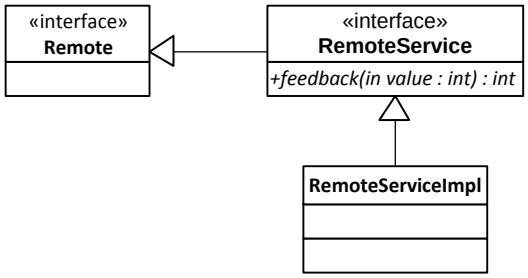
More details about project can always be found from http://hoangtuanonline.com/qthessian/

4.4.4 Instruction for the user to create a simple Spring remote service with STS

From STS, the user needs to create a new Spring MVC template project by New → Spring Template Project → Spring MVC Project. If this is the first time creating the project, it will take some minutes for STS to download the project template from the Internet. Then the user will be asked to enter project name and the default package.

When the project template is downloaded, user should navigate to the created projects and allocate the pom.xml file. This file contains the dependencies managed by maven. The org.springframework-version should be 3.0.5.BUILD-RELEASE, because this will be used as default version for all Spring dependencies. By adding the Hessian's dependency to the dependencies list, maven will automatically manage and download the Hessian library (See Appendix 5).

The next step involves creating a remote service interface and implements that service:



GRAPH 23. Remote Service

The final step comprises of registering the service implementation and remote interface with the Spring framework. By registering the beans and the service interface with HessianServiceExport, the service is now accessible by any Hessian clients with the mapping path /Test. In localhost, for example, it is "http://localhost:8080/TestService/Test"

4.4.5 Service access from Qt

In order to use our implementation, it is necessary to add network nature to the Qt project configuration file and include the QtHessian header ("QtHessian.h")

After that, a network manager must be created to represent the service exported in the Chapter 5.4 with a url. The url shows where the service is located, in this example, the url must end with with *"/Test"*. If the user runs the service from Chapter 5.4 on local host, the url will be: http://localhost:8080/<project name>/Test

4.5 Sample test cases of the project

Test cases were created using remote method with no parameter which is described in Table 13 and Table 20). The test with integer is shown in Table 14 and Table 21, while the test with double is demonstrated in Table 15 and Table 22. Next, Table 16 and Table 23 show the test with long value. The string test will be provided in Table 17 and Table 24. The DateTime format test is shown in Table 18 and Table 25, and the test of binary values is demonstrated in Table 19 and Table 26. The cases were tested in both synchronous and asynchronous mode.

4.5.1 Test1: remote method with no parameter

TABLE 13. Test with no parameter

Purpose	Test that developer can make simple RPC call without any input parameters	
Prereq		
Test data	Method name	hello
Steps	1. Developer initialize necessary objects 2. Developer make call 3. Developer verify the return pattern	
Note	Return pattern is defined inside the interface and implementation. It is not a null	

4.5.2 Test2: remote method with Integer

TABLE 14. Test with integer value

Purpose	Test that developer can make simple RPC call with integer	
Prereq		
Test data	Method name	feedback
	value	boundary values as specified in the protocol
		random values generated from random generator
Steps	1. Developer initialize necessary objects 2. Developer create the call and serialize the data 3. Developer make call 4. Developer verify the return pattern with the sent pattern	
Note		

4.5.3 Test3: remote method with Double

TABLE 15. Test with double value

Purpose	Test that developer can make simple RPC call with double	
Prereq		
Test data	Method name	feedbackDouble
	value	boundary values as specified in the protocol
		random values generated from random generator
Steps	1. Developer initialize necessary objects 2. Developer create the call and serialize the data 3. Developer make call 4. Developer verify the return pattern with the sent pattern	
Note	Double can’t be compared using “==” operator	

4.5.4 Test4: remote method with Long

TABLE 16. Test with long value

Purpose	Test that developer can make simple RPC call with integer 64 bit	
Prereq		
Test data	Method name	feedbackLong

	value	boundary values as specified in the protocol
		random values generated from random generator
Steps	1. Developer initialize necessary objects 2. Developer create the call and serialize the data 3. Developer make call 4. Developer verify the return pattern with the sent pattern	
Note		

4.5.5 Test5: remote method with DateTime

TABLE 17. Test with DateTime value

Purpose	Test that developer can make simple RPC call with DateTime type	
Prereq		
Test data	Method name	feedbackDateTime
	value	today
		random days generated by random generator
Steps	1. Developer initialize necessary objects 2. Developer create the call and serialize the data 3. Developer make call 4. Developer verify the return pattern with the sent pattern	
Note	DateTime is encoded as a long (integer64)	

4.5.6 Test6: remote method with String

TABLE 18. Test with String value

Purpose	Test that developer can make simple RPC call with String	
Prereq		
Test data	Method name	feedbackString
	value	random string with length as specified in the protocol
		random string length generated by generator
Steps	1. Developer initialize necessary objects 2. Developer create the call and serialize the data 3. Developer make call 4. Developer verify the return pattern with the sent pattern	
Note	String can’t be compared using == operator UTF-8 String length is limited to 2^16.	

4.5.7 Test7: remote method with Binary

TABLE 19. Test with binary value

Purpose	Test that developer can make simple RPC call with binary data	
Prereq		
Test data	Method name	feedbackBinary
	value	random data with boundary length specified in the protocol
		random generated length data with random data
Steps	1. Developer initialize necessary objects 2. Developer create the call and serialize the data 3. Developer make call 4. Developer verify the return pattern with the sent pattern	
Note		

4.5.8 Test8: remote method with no parameter in asynchronous

TABLE 20. Test with no parameter

Purpose	Test that developer can make simple RPC call without any input parameters in asynchronous mode	
Prereq		
Test data	Method name	hello
Steps	1. Developer initialize necessary objects 2. Developer make call 3. Developer verify the return pattern	
Note	Return pattern is defined inside the interface and its implementation. It is not a null	

4.5.9 Test9: remote method with Integer in asynchronous mode

TABLE 21. Test with integer

Purpose	Test that developer can make simple RPC call with integer in asynchronous mode	
Prereq		
Test data	Method name	feedback
	value	boundary values as specified in the protocol
		random values generated from random generator
Steps	1. Developer initialize necessary objects 2. Developer create the call and serialize the data 3. Developer make call 4. Developer verify the return pattern with the sent pattern	
Note		

4.5.10 Test10: remote method with Double in asynchronous mode

TABLE 22. Test with double

Purpose	Test that developer can make simple RPC call with double in asynchronous mode	
Prereq		
Test data	Method name	feedbackDouble
	value	boundary values as specified in the protocol
		random values generated from random generator
Steps	1. Developer initialize necessary objects 2. Developer create the call and serialize the data 3. Developer make call 4. Developer verify the return pattern with the sent pattern	
Note	Double can’t be compared using “==” operator	

4.5.11 Test11: remote method with Long in asynchronous mode

TABLE 23. Test with long

Purpose	Test that developer can make simple RPC call with integer 64 bit in asynchronous mode	
Prereq		
Test data	Method name	feedbackLong
	value	boundary values as specified in the protocol
		random values generated from random generator

Steps	1. Developer initialize necessary objects 2. Developer create the call and serialize the data 3. Developer make call 4. Developer verify the return pattern with the sent pattern
Note	

4.5.12 Test12: remote method with DateTime in asynchronous mode

TABLE 24. Test with DateTime

Purpose	Test that developer can make simple RPC call with DateTime type in asynchronous mode	
Prereq		
Test data	Method name	feedbackDateTime
	number	today
		random days generated by random generator
Steps	1. Developer initialize necessary objects 2. Developer create the call and serialize the data 3. Developer make call 4. Developer verify the return pattern with the sent pattern	
Note	DateTime is encoded as a long (integer64)	

4.5.13 Test13: remote method with String in asynchronous mode

TABLE 25. Test with String

Purpose	Test that developer can make simple RPC call with String in asynchronous mode	
Prereq		
Test data	Method name	feedbackString
	value	random string with length as specified in the protocol
		random string length generated by generator
Steps	1. Developer initialize necessary objects 2. Developer create the call and serialize the data 3. Developer make call 4. Developer verify the return pattern with the sent pattern	

Note	String can't be compared using == operator UTF-8 String length is limited to 2^16

4.5.14 Test14: remote method with Binary in asynchronous mode

TABLE 26. Test with binary

Purpose	Test that developer can make simple RPC call with binary data in asynchronous mode	
Prereq		
Test data	Method name	feedbackBinary
	value	random data with boundary length specified in the protocol
		random generated length data with random data
Steps	1. Developer initialize necessary objects 2. Developer create the call and serialize the data 3. Developer make call 4. Developer verify the return pattern with the sent pattern	
Note		

4.6 Performance of test cases

In this part, the tables are made to compare the performance of system between Qt platform and J2SE.

4.6.1 Test 15: Qt vs Java (J2SE) performance

The test of arithmetic performance is shown in Table 27. It also describes the test data as well as the steps for execution.

TABLE 27. Arithmetic performance

Purpose	Test arithmetic performance of C++(Qt) vs Java (J2SE)
Prereq	Quad-core AMD 9850 2.50GHz 4GB RAM

Test data	The core algorithm is showed as below:
	`for ( i = 0; i < 4000; i ++)`
	`for ( j=0; j < 1000; j++)`
	`res = j*i;`

Steps	The algorithm is repeated 100 times, 500 times and 1000 times. The test is executed 5 times and the result is the average of the 5 runs.
Note	Time of execution must be recorded and averaged

4.6.2 Test 16: Qt vs Java (J2SE) integer serialization performance

The test of interger serialization performance is shown in Table 28. It also provides the test data as well as the steps for execution.

TABLE 28. Integer serialization performance

Purpose	Test integer serialization performance of C++(Qt) vs Java (J2SE)	
Prereq	Quad-core AMD 9850 2.50GHz 4GB RAM	
Test data	value	random integers generated by random generator
Steps	We create a test of sending and receiving 1000, 10.000 and 40,000 integer numbers. The test, again, is executed 5 times for both Qt and Java, and then the average will be taken to get the final result.	
Note	Time of execution must be recorded and averaged	

5 SUMMARY OF RESULTS

The test results indicated that our test cases either met or exceeded the requirements specified in each case. On the performance test, executing times of each run was recorded, statically analyzed and compared with the corresponding Java execution. Details result can be found as follows:

5.1 Unit test

The parser unit and the serialization unit were tested intensively with a set of boundary values and random sample values collected by analyzing Hessian Java network frames and from Hessian documentation. All possible values that would generate errors were tested to ensure their reliability.

Then the parser and the serialization unit were tested with a memory stream and a text stream with boundary values, random values and sequential random values. Data was first serialized by the serialization unit and written to the stream, and then the parser unit read the stream and reconstructed the data. The income and outcome data was checked if they matched in the binary level.

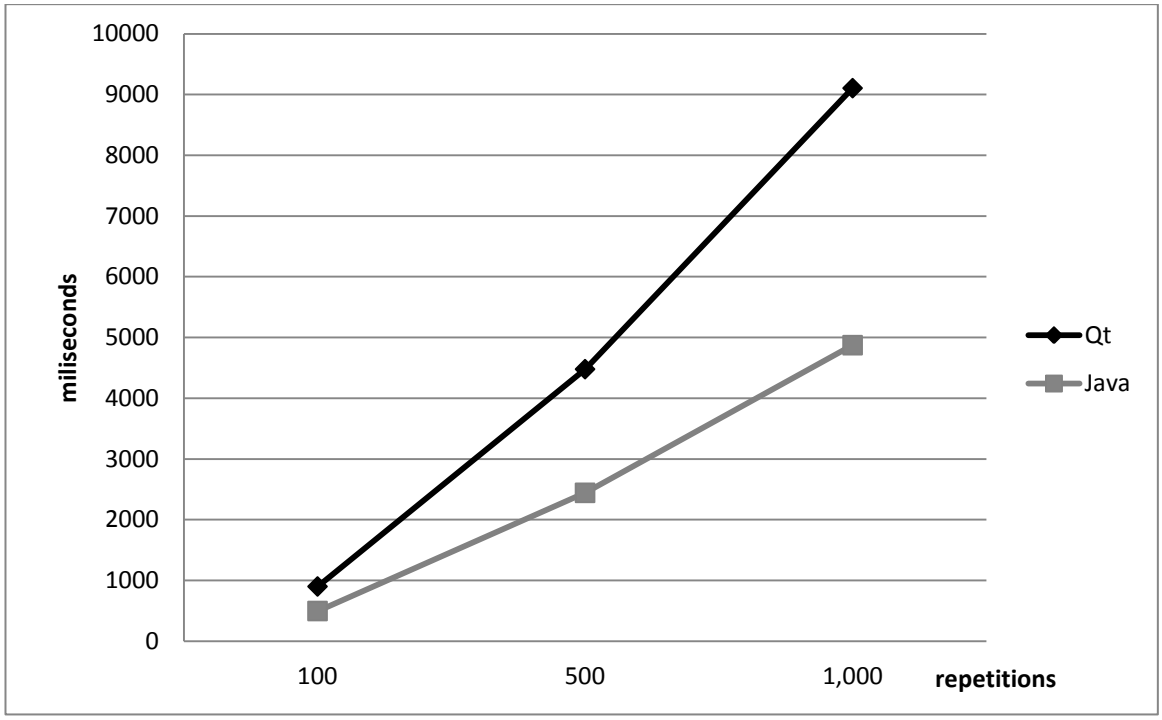
Upon satisfactory completion of the above test, the two units were used in the integration tests with the network unit. A test case consisted of boundary numbers, random generated numbers and sequential numbers are serialized and send to the network unit, the network unit connects and received the numbers from feedback service. The received packets were parsed by the parser unit before they were compared with the input values.

5.2 Performance test

We built performance tests between Qt and Java program on a Quad-core AMD 9850 2.50GHz computer.

5.2.1 Test 15: Qt vs Java (J2SE) performance

Result



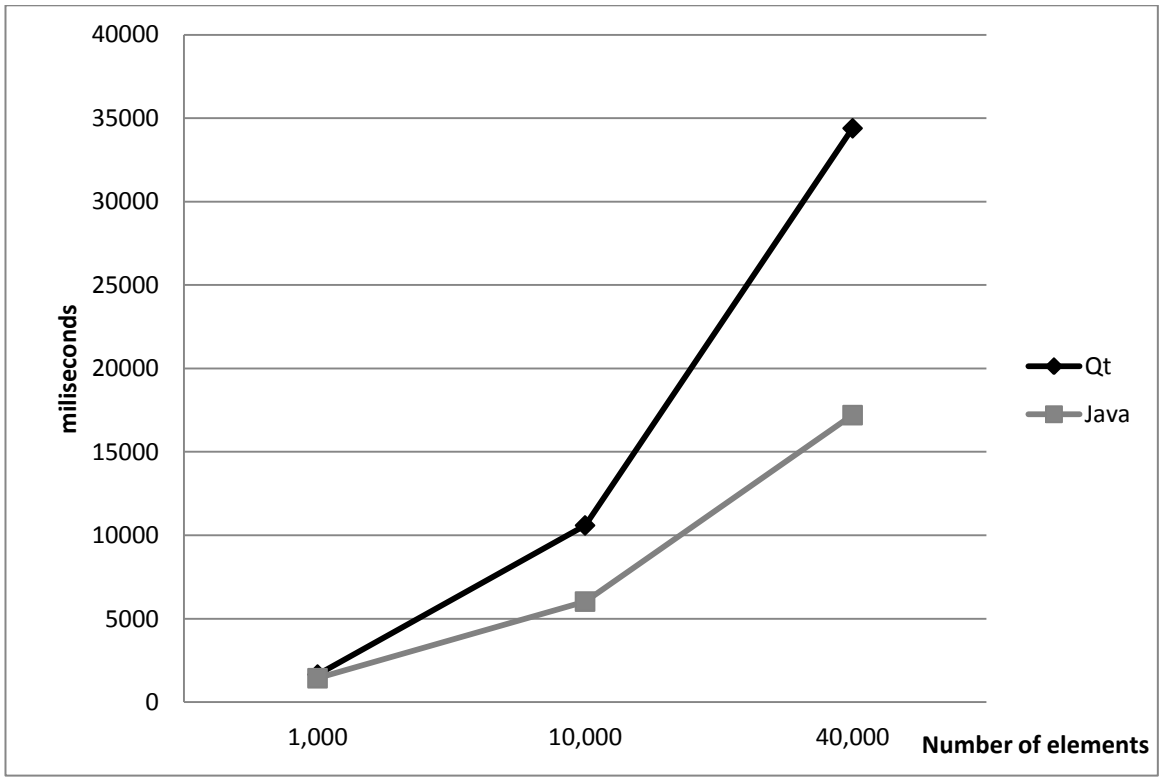
GRAPH 24. Test for iterating and multiplication

Comment

It is surprising that Java outperforms Qt in the iterating and multiplication test. In most of the test, the Qt consumes twice of the time Java needed to complete the test. By doing further analysis, the JVM is acknowledged to make use of two processor cores while the Qt's program runs on a single core.

5.2.2 Test 16: Qt vs Java (J2SE) integer serialization performance

Result



GRAPH 25. Feedback integer test.

Comment

Our Qt implementation does not fare quite well in this test; even if it performs quite close to the Java client on small numbers. However, compared to the result achieved from the test 1, this result means that Qt implementation has similar performance with Java implementation.

6 CONCLUSION

This thesis presented the design and partial implementation of Hessian 2.0 Protocol on Qt platform. The implementation was designed to be used in cross-platform Qt applications

that require interactions with web services. By implementing the protocol in Qt, a cross-platform UI framework, the developer will be able to implement Web-service-enabled application and deploy across desktop, mobile and embedded OS without rewriting the source code.

In the implementation, we introduced HCall as the local representation of remote methods. HCall simplifies remote method access, making it as easy as accessing local one. Furthermore, we implemented our custom HNetworkManager class that extends the Qt Network module to support both synchronous and asynchronous mode. In the academic context, we limit our implementation to eight primitive data types and do intensive testing on these cases. This makes the implementation reliable yet flexible so that it can be expanded in many ways. In addition, we also provided a simple way to implement custom serialization for complex object types. Therefore, developers will find our work fitting a variety of uses. The implementation was also carefully tested with Qt built-in unit testing, integration and performance testing.

The current implementation is only applied to conventional messing and RPC communication pattern because these two patterns are commonly used as web service protocol. Certainly, the implementation could be expanded to support streaming mechanism. Other improvements may include a custom encryption-decryption algorithm to improve security, a compression mechanism to save the bandwidth and improve application throughput, a caching algorithm combined with mapping of types, values, classes to improve general efficiency.

REFERENCES

Arnold, K, Gosling, J. and Homes, D. 2000. The Java Programming Language (3rd Edition). Addison Wesley.

Baker, S. 1994. COBRA Implementation Issues. IEE Colloquium on Distributed Object Management.

Campione, M. and Walrath, K. 1998. The Java Tutorial (2nd Edition).

CauchoTechnology. 2007a. Hessian 2.0 Serialization Protocol. Available at: http://hessian.caucho.com/doc/hessian-serialization.html Accessed 25 November, 2010.

CauchoTechnology. 2007b. Hessian 2.0 Web Services Protocol. Available at: http://hessian.caucho.com/doc/hessian-ws.html Accessed 1 December, 2010

CauchoTechnology. 2007c. Hessian Binary Web Service Protocol. Available at: http://hessian.caucho.com/#Tests Accessed 25 November, 2010.

CauchoTechnology. 2007d. Burlap. Available at: http://hessian.caucho.com/doc/burlap.xtp Accessed 1 December, 2010.

Gredler, D. 2008. Java Remoting Benchmarks. Available at: http://daniel.gredler.net/2008/01/07/java-remoting-protocol-benchmarks/. Accessed 10 December, 2010

Harrop, R. and Machacek, J. 2005. Pro Spring. Apress.

Heather, K. 2001. Web Service Conceptual Architecture. IBM.

Inc., Caucho Technology 2005. Caucho Technology and AldraTech announce C++ Hessian functionality. Caucho Press

Ingham, D., Rees, O. and Norman, A. 1999. CORBA transactions through firewalls. International Symposium Distributed Objects and Applications.

NokiaQt. QNetworkAccessManager Class Reference v4.7. Available at: http://doc.trolltech.com/4.7/qnetworkaccessmanager.html Accessed at 1 December, 2010.

NokiaQt. QNetworkReply Class Reference v4.7. Available at: http://doc.trolltech.com/4.7/qnetworkreply.html Accessed 1 December, 2010.

NokiaQt. QNetworkRequest Class Reference v4.7. Available at: http://doc.trolltech.com/4.7/qnetworkrequest.html Accessed 1 December, 2010.

NokiaQt. QVariant Class Reference. Available at: http://doc.qt.nokia.com/latest/qvariant.html Accessed 1 December, 2010.

Paul, S., Santiago, P., Kohuske, K., Marc, H. and Eduardo, P. 2003. Fast Web Service. Sun Developer Network. Oracle.

Pitt, E. and McNiff, K. 2001. java.rmi: The Remote Method Invocation. Addison Wesley Professional.

Sharp, R. 2008. Principle of protocol design. Lyngby: Springer.

Springsource. Remoting and webservices using Spring. Available at: http://static.springsource.org/spring/docs/2.0.x/reference/remoting.html Accessed 1 December, 2010.

Van de Velde, T., Snyder, B., Dupuis, C., Li, S., Horton, A. and Balani, N. 2007. Beginning Spring Framework 2.0. Indianapolis, Indiana: Wiley Publishing Inc.

Walton, C. 2005. Protocols for Web Service Invocation. School of Informatics, University of Edinburg, UK.

Wei, G. 2006. Research and Development of Interprocess Communication Technology of Distributed System. Beijing: Beijing University of Technology

REMOTE METHOD INVOCATION (RMI)

Definition

Remote method invocation is the invocation of a method in a remote object via a remote interface. The remote interface must extend *java.rmi.Remote* to define the methods that will be available remotely.

Characteristics

Table below show a comparison between local method and remote method:

TABLE 29. Argument passing

Type	**Local method**	**Remote method**
Primitive types	By value	By value
Object	Be reference	By value(deep copy)
Exported remote object	By reference	By remote reference

In addition, the remote method also has the following semantics:

- It can only be invoked via a remote interface which declares it.
- It must throw remote exception.

— Clients must catch and deal with remote exception.
— The semantics of *java.lang.Object* are specialized for remote objects.

RMI generally implements either “at least once” or “at most once” semantics. “At least once” system makes sure that the remote procedure has executed at least once, possibly more than once, while the “at most once” guarantees that the procedure has executed exactly once. Therefore, a remote method which throws a remote exception may or may not have executed it at all.

Since the remote method involves two or more computers, there is a possibility that one may fail while others do not. This is considered as partial failure. Moreover, the computers have their own memory space and cannot access the memory of others. This is the reason why RMI arguments and results are passed by “deep copy” rather than by reference. Communication over networks can have failure and data may never arrive. For this reason, a “wait timeout” should be performed after dispatching RMI, rather than an indefinite wait. Due to these characteristics, a RMI system must be carefully designed and built.

Remote interfaces

A remote interface must satisfy the following conditions:

— It must extend *java.rmi.Remote* (this interface extends no interface and exports no method. It is an interface which distinguishes remote interfaces from non-remote interfaces).
— Every method which it exports, either explicitly or by inheritance, must declare that it throws *RemoteException*.

— When a remote object can be marshaled as a parameter or result of a remote method, it must be declared as its remote interface, not its actual implementation class.

RMI proxies are the mechanism on client side, while the corresponding mechanism on the server side is called the dispatcher. A dispatcher mediates between the RMI run-time and the corresponding remote object: it receives the call packet and dispatches the call to the remote object.

RMI clients

RMI clients acquire objects, invoke method on objects, use the results and catch the exceptions thrown by the methods. The method invocation may fail completely or partially. For instance, the remote server may have completely succeeded in committing a database transaction, but then have experienced a failure when transmitting a result parameter.

On the other hand, clients of remote objects must deal with latency. The latency is dependent on: network bandwidth, network delay, server capacity, server load, method execution and etc…

(Pitt and McNiff. 2001, 46-48.)

HESSIAN GRAMMAR

```
# starting production
top        ::= value

# 8-bit binary data split into 64k chunks
binary     ::= x41 b1 b0 <binary-data> binary # non-final chunk
           ::= 'B' b1 b0 <binary-data>        # final chunk
           ::= [x20-x2f] <binary-data>        # binary data of
                                              #  length 0-15
           ::= [x34-x37] <binary-data>        # binary data of
                                              #  length 0-1023

# boolean true/false
boolean    ::= 'T'
           ::= 'F'

# definition for an object (compact map)
class-def  ::= 'C' string int string*

# time in UTC encoded as 64-bit long milliseconds since epoch
date       ::= x4a b7 b6 b5 b4 b3 b2 b1 b0
           ::= x4b b3 b2 b1 b0       # minutes since epoch

# 64-bit IEEE double
double     ::= 'D' b7 b6 b5 b4 b3 b2 b1 b0
           ::= x5b                   # 0.0
           ::= x5c                   # 1.0
           ::= x5d b0                # byte cast to double
                                     #  (-128.0 to 127.0)
           ::= x5e b1 b0             # short cast to double
           ::= x5f b3 b2 b1 b0       # 32-bit float cast to double

# 32-bit signed integer
int        ::= 'I' b3 b2 b1 b0
           ::= [x80-xbf]             # -x10 to x3f
           ::= [xc0-xcf] b0          # -x800 to x7ff
           ::= [xd0-xd7] b1 b0       # -x40000 to x3ffff

# list/vector
list        ::= x55 type value* 'Z'   # variable-length list
```

```
            ::= 'V' type int value*   # fixed-length list
            ::= x57 value* 'Z'        # variable-length untyped list
            ::= x58 int value*        # fixed-length untyped list
            ::= [x70-77] type value*  # fixed-length typed list
            ::= [x78-7f] value*       # fixed-length untyped list

# 64-bit signed long integer
long       ::= 'L' b7 b6 b5 b4 b3 b2 b1 b0
           ::= [xd8-xef]              # -x08 to x0f
           ::= [xf0-xff] b0           # -x800 to x7ff
           ::= [x38-x3f] b1 b0        # -x40000 to x3ffff
           ::= x59 b3 b2 b1 b0        # 32-bit integer cast to long

# map/object
map         ::= 'M' type (value value)* 'Z'  # key, value map pairs
            ::= 'H' (value value)* 'Z'       # untyped key, value

# null value
null       ::= 'N'

# Object instance
object     ::= 'O' int value*
           ::= [x60-x6f] value*

# value reference (e.g. circular trees and graphs)
ref        ::= x51 int            # reference to nth map/list/object

# UTF-8 encoded character string split into 64k chunks
string     ::= x52 b1 b0 <utf8-data> string  # non-final chunk
           ::= 'S' b1 b0 <utf8-data>         # string of length
                                             #  0-65535
           ::= [x00-x1f] <utf8-data>         # string of length
                                             #  0-31
           ::= [x30-x34] <utf8-data>         # string of length
                                             #  0-1023

# map/list types for OO languages
type       ::= string                        # type name
           ::= int                           # type reference

# main production
```

```
value      ::= null
           ::= binary
           ::= boolean
           ::= class-def value
           ::= date
           ::= double
           ::= int
           ::= list
           ::= long
           ::= map
           ::= object
           ::= ref
           ::= string
```

Hessian is organized by a bytecode protocol. Here is the bytecode encoding:

```
      x00 - x1f    # utf-8 string length 0-32
      x20 - x2f    # binary data length 0-16
      x30 - x33    # utf-8 string length 0-1023
      x34 - x37    # binary data length 0-1023
      x38 - x3f    # three-octet compact long (-x40000 to x3ffff)
      x40          # reserved (expansion/escape)
      x41          # 8-bit binary data non-final chunk ('A')
      x42          # 8-bit binary data final chunk ('B')
      x43          # object type definition ('C')
      x44          # 64-bit IEEE encoded double ('D')
      x45          # reserved
      x46          # boolean false ('F')
      x47          # reserved
      x48          # untyped map ('H')
      x49          # 32-bit signed integer ('I')
      x4a          # 64-bit UTC millisecond date
      x4b          # 32-bit UTC minute date
      x4c          # 64-bit signed long integer ('L')
      x4d          # map with type ('M')
      x4e          # null ('N')
      x4f          # object instance ('O')
      x50          # reserved
      x51          # reference to map/list/object - integer ('Q')
```

```
x52          # utf-8 string non-final chunk ('R')
x53          # utf-8 string final chunk ('S')
x54          # boolean true ('T')
x55          # variable-length list/vector ('U')
x56          # fixed-length list/vector ('V')
x57          # variable-length untyped list/vector ('W')
x58          # fixed-length untyped list/vector ('X')
x59          # long encoded as 32-bit int ('Y')
x5a          # list/map terminator ('Z')
x5b          # double 0.0
x5c          # double 1.0
x5d          # double represented as byte (-128.0 to 127.0)
x5e          # double represented as short (-32768.0 to 327676.0)
x5f          # double represented as float
x60 - x6f    # object with direct type
x70 - x77    # fixed list with direct length
x78 - x7f    # fixed untyped list with direct length
x80 - xbf    # one-octet compact int (-x10 to x3f, x90 is 0)
xc0 - xcf    # two-octet compact int (-x800 to x7ff)
xd0 - xd7    # three-octet compact int (-x40000 to x3ffff)
xd8 - xef    # one-octet compact long (-x8 to xf, xe0 is 0)
xf0 - xff    # two-octet compact long (-x800 to x7ff, xf8 is 0)
```

Here is the Hessian message bytecode map:

```
x00 - x42    # reserved
x43          # rpc call ('C')
x44          # reserved
x45          # envelope ('E')
x46          # fault ('F')
x47          # reserved
x48          # hessian version ('H')
x49 - x4f    # reserved
x4f          # packet chunk ('O')
x50          # packet end ('P')
x51          # reserved
x52          # rpc result ('R')
```

```
x53 - x59    # reserved
x5a          # terminator ('Z')
x5b - x5f    # reserved
x70 - x7f    # final packet (0 - 4096)
x80 - xff    # final packet for envelope (0 - 127)
```

The Hessian classes can be used for serialization and de-serialization. Hessian's object serialization has 8 primitive types:

Raw binary data

```
binary ::= b b1 b0 <binary-data> binary
       ::= B b1 b0 <binary-data>
       ::= [x20-x2f] <binary-data>
```

Binary data is encoded in chunks. The octet x42('B') encodes the final chunk and x62('b') represents any non-final chunk. Each chunk has 16-bit length value (length = 256 * b1 + b0). However, binary data with length less than 15 can be encoded by a single octet length [x20-x2f] (length = code – 0x20)

Example:

```
x20               # zero-length binary data
x23 x01 x02 x03   # 3 octet data
B x10 x00 ....    # 4k final chunk of data
b x04 x00 ....    # 1k non-final chunk of data
```

Boolean

```
boolean ::= T
        ::= F
```

The octet ‘F’ is for false and octet ‘T’ for true.

64-bit millisecond date

```
date ::= x4a b7 b6 b5 b4 b3 b2 b1 b0
     ::= x4b b4 b3 b2 b1 b0
```

— It is represented by a 64-bit long of millisecond since Jan 1, 1970 00:00h, UTC.

— The second form contains a 32-bit int of minutes since Jan 1, 1970 00:00h, UTC

Example:

```
x4a x00 x00 x00 xd0 x4b x92 x84 xb8   # 09:51:31 May 8, 1998 UTC
x4b x4b x92 x0b xa0                  # 09:51:00 May 8, 1998 UTC
```

64-bit double

```
double ::= D b7 b6 b5 b4 b3 b2 b1 b0
       ::= x5b
       ::= x5c
       ::= x5d b0
       ::= x5e b1 b0
       ::= x5f b3 b2 b1 b0
```

— This is a 64-bit IEEE floating point number.

— The double 0.0 can be represented by the octet x5b, while double 1.0 is represented by x5c.

— Double between -128.0 and 127.0 (no fraction component) can be represented in two octets by casting the byte value to a double, i.e.

Value = (double) b0

— Double between -32768.0 and 32767.0 (no fraction component) can be represented in three octets by casting the short value to a double, i.e.

Value = (double) (256*b1 + b0)

— Doubles which are equivalent to their 32-bit float representation can be represented as the 4-octet float and then cast to double.

Example:

```
D x40 x28 x80 x00 x00 x00 x00 x00  # 12.25
```

32-bit int

```
int ::= 'I' b3 b2 b1 b0
    ::= [x80-xbf]
    ::= [xc0-xcf] b0
    ::= [xd0-xd7] b1 b0
```

— An integer is represented by the octet x49 (‘I’) followed by the 4 octets of the integer in big-endian order, i.e.

Value = (b3 << 24) + (b2 << 16) + (b1 << 8) + b0

— Integers between -16 and 47 can be encoded by a single octet in the range x80 to xbf, i.e.

Value = code – 0x90

— Integers between -2048 and 2047 can be encoded in two octets with the leading byte in the range xc0 to xcf, i.e.

Value = ((code – 0xc8) << 8) + b0

— Integers between -262144 and 262143 can be encoded in three bytes with the leading byte in the range xd0 to xd7, i.e.

Value = ((code – 0xd4) << 16) + (b1 << 8) + b0

Example:

```
I x00 x00 x00 x00  # 0
I x00 x00 x01 x2c  # 300
```

64-bit long

```
long ::= L b7 b6 b5 b4 b3 b2 b1 b0
     ::= [xd8-xef]
     ::= [xf0-xff] b0
     ::= [x38-x3f] b1 b0
     ::= x59 b3 b2 b1 b0
```

— A long is represented by the octet x59 (‘L’) followed by the 8-bytes of the integer in big-endian order.

— Long between -8 and 15 are represented by a single octet in the range xd8 to xef, i.e.

Value = code – 0xe0

— Long between -2048 and 2047 are encoded in two octets with leading byte in the range xf0 to xff, i.e.

Value = ((code – 0xf8) << 8) + b0

— Long between -262144 and 262143 are encoded in three octets with leading byte in the range x38 to x3f, i.e.

Value = ((code – 0x3c) << 16) + (b1 << 8) + b0

— Longs which fit into 32-bits are encoded in five octets with the leading byte x4c, i.e.

Value = (b3 << 24) + (b2 << 16) + (b1 << 8) + b0

Example:

```
L x00 x00 x00 x00 x00 x00 x01 x2c  # 300
```

Null

```
null ::= N
```

Null represents a null pointer. The octet 'N' represents a null value.

UTF8-encoded string

```
string ::= x52 b1 b0 <utf8-data> string
       ::= S b1 b0 <utf8-data>
       ::= [x00-x1f] <utf8-data>
       ::= [x30-x33] b0 <utf8-data>
```

— It is a 16-bit Unicode character string encoded in UTF-8. x53 ('S') represents a final chunk while x52('R') represents any non-final chunk. Each chunk has a 16-bit unsigned integer length value. String chunk may not split surrogate pairs.

— String with length less than 32 maybe encoded with a single octet length [x00-x1f], i.e.

Value = code

Example:

```
x00                 # "", empty string
x05 hello           # "hello"
x01 xc3 x83         # "\u00c3"
S x00 x05 hello     # "hello" in long form
```

Three recursive types:

List for lists and arrays

```
list ::= x55 type value* 'Z'   # variable-length list
     ::= 'V' type int value*   # fixed-length list
     ::= x57 value* 'Z'        # variable-length untyped list
     ::= x58 int value*        # fixed-length untyped list
     ::= [x70-77] type value*  # fixed-length typed list
     ::= [x78-7f] value*       # fixed-length untyped list
```

— An ordered list is like an array. Two list productions are fixed-length lists. Both lists have a type; and the type string may have an arbitrary UTF-8 string understood by a service.

— Any parser expecting a list must also accept a null or a shared ref. The valid value of type depends on specific application.

— Hessian 2.0 allows a compact form of a list for successive list of the same type where the length is known.

Example:

Fixed length type

```
x72                # typed list length=2
x04 [int         # type for int[] (save as type #0)
x90              # integer 0
x91              # integer 1
```

Map for maps and dictionaries.

map ::= M type (value value)* Z

It represents serialized maps and objects. The type element describes the type of map. The type can be empty, i.e. a zero length. If the type is not specified, the parser is responsible for choosing it. For objects, unrecognized keys will be ignored. Any time the parser expects a map, it must be able to support a null or ref.

Example: Map representation of a Java Object

```
public class Car implements Serializable {
  String color = "aquamarine";
  String model = "Beetle";
  int mileage = 65536;
}

M
  x13 com.caucho.test.Car  # type
  x05 color                # color field
  x0a aquamarine
  x05 model                # model field
  x06 Beetle
  x07 mileage              # mileage field
  I x00 x01 x00 x00
  Z
```

Object for objects.

```
class-def  ::= 'C' string int string*
object     ::= 'O' int value*
           ::= [x60-x6f] value*
```

- Hessian 2.0 has a compact object form where the field names are serialized once. Following objects only need to serialize their values.
- The object definition (type string, number of fields, field names) is stored in the object definition map.
- The object instantiation creates new object based on previous definition.

Example: Object serialization

```
class Car {
  String color;
  String model;
}
out.writeObject(new Car("red", "corvette"));
out.writeObject(new Car("green", "civic"));
---
C                        # object definition (#0)
  x0b example.Car        # type is example.Car
  x92                    # two fields
  x05 color              # color field name
  x05 model              # model field name
O                        # object def (long form)
  x90                    # object definition #0
  x03 red                # color field value
  x08 corvette           # model field value
```

One special construct in Hessian is:

ref for shared and circular object references.

```
ref ::= x51 int
```

- It is an integer referring to a previous list, map or object instance. In each list, map or object is read from input stream. Ref can refer to incompletely-read items.
- Each map or array is stored into an array as it is parsed. Ref selects one of the stored object. The first object is numbered ‘0’
- Ref only refers to list, map and object elements.

Example of circular list:

```
list = new LinkedList();
list.data = 1;
list.tail = list;
---
C
  x0a LinkedList
  x92
```

```
x04 head
x04 tail
x90      # object stores ref #0
x91      # data = 1
x51 x90  # next field refers to itself, i.e. ref #0
```

In addition, Hessian also has three internal reference maps:

1. An *object/list* reference map
2. A class definition reference map
3. A type (classname) reference map

— The value reference map let Hessian support arbitrary map by adding list, object and map as it encounter them in bytecode stream. Hessian also supports recursive and circular data structure.
— Hessian efficiency is improved by avoiding repetition of common string data.
— For class reference, each object definition is automatically added to class-map.
— For type reference, the type strings for map and list values are stored in the type map for reference.

TABLE 30. Mapping of Qt-Java types

Qt	**Hessian**	**Java**
QVariant()	null	null
bool	boolean	boolean, Boolean
qint32	32-bit int	Integer, int
qint64	64-bit int	Long, long
double	double	Double, double
QString	string	String, string
QDateTime	64-bit int	Date
QByteArray	raw binary data	byte[]
QList	list	ArrayList
QMap	map	HashMap

TABLE 31. Mapping of Java-Qt types

Java	**Hessian**	**Qt**
null	null	QVariant()
Boolean/Boolean	bool	QVariant(bool)
byte/Byte	32-bit int	QVariant(qint32)
short/Short	32-bit int	QVariant(qint32)
int/Int	32-bit int	QVariant(qint32)
long/Long	64-bit int	QVariant(qint64)
float	double	QVarian(double)
double	double	QVarian(double)
char/char[]/string/String	string	QVariant(QString)
Date	64-bit long	QVariant(QDateTime)
byte[]	raw binary data	QVariant(QByteArray)
List	list	QVariant(QList)
Vector	list	QVariant(QList)
Map	map	QVariant(QMap)
HashMap	map	QVariant(QMap)
Hashtable	map	QVariant(QMap)

SPRING FRAMEWORK

Introduction

Spring began its life as an alternative to the heavyweight J2EE 1.4 containers. Its first appearance was in the sample code that Rod Johnson wrote in his 2002 Wrox Press book "Expert One on One Java J2EE Design and Development". Rod's description was a breath of fresh air to the J2EE development community. The introduced lightweight container can minimize the complexity of a server-side application construction. Rod and his group continued to develop the framework, so the adoption of the framework continues to spread worldwide. By 2007, version 2 of Spring Framework was released and version 3 was released by the late 2009.

Spring enables a construction of completely component-based applications. The core of Spring provides flexible runtime wiring of Java Beans via an XML-based descriptor. Applications can be created by wiring together JavaBeans with Spring supplied container service components.

In Spring, containers and beans form the backbone of your application. A bean is an object instantiated, assembled and managed by Spring IoC (Inversion of Control) Container. IoC is used throughout Spring application to decouple dependencies that typically increase complexity in J2EE environment. Aspect-Oriented Programming (AOP) is a core enabler in Spring. It enables you to factor cross-cutting concerns out of your application and maintain them separately from the body of code.

Spring Remoting

Remoting is defined as the exposing of code for remote access without having to write another layer of software around the code to be exposed. Spring uses various technologies for remoting support to ease the development of remote-enabled service. Currently, Spring supports the following technologies (Remoting and Web Services using Spring, 2010):

- Remote Method Invocation (RMI): by the use of *RmiProxyFactoryBean* and the *RmiServiceExporter* Spring support both traditional RMI (with *java.rmi.Remote* interfaces and *java.rmi.RemoteException*) and transparent remoting via RMI invokers.
- Spring's HTTP invoker: Spring provides a special remoting strategy with allow for Java serialization via HTTP, supporting any Java interface. The support classes are *HttpInvokerProxyFactoryBean* and *HttpInvokerServiceExporter*.
- Hessian: by using the *HessianProxyFactoryBean* and the *HessianServiceExporter* to transparently expose your services using binary protocol provided by Caucho.
- Burlap: Spring support Caucho XML-based exposing of service by using *BurlapProxyFactoryBean* and *BurlapServiceExporter*.
- JAXRPC: Spring provides remoting support for web services via JAX-RPC.

QT PLATFORM

Qt Classes

The Qt C++ class library provides a rich set of functionality needed to build advanced, cross-platform applications. The library is intuitive, easy to use and learn, and producing highly readable and easily maintainable code. The thesis makes use of the QtNetwork package which was designed especially for handing multiple network connection simultaneously.

QtNetwork package

QNetworkAccessManager class

This class is the core of the package, allowing the application to send requests and receive replies. The *QNetworkAccessManager* object, which holds the common configurations and settings for the requests it sends, is created. Qt Reference Document suggests that one object should be enough for the whole Qt application (QNetworkAccessManager Class Reference , 2010). Once the object was created, the application can use it to send requests over network. The returned object, the *QNetworkReply* object, is used to obtain any data returned in response to the corresponding request.

With the help of the Bearer Management API to Qt 4.7, *QNetworkAccessManager* gained the ability to manage network connections. It can start the network interface if the device is offline and terminates the interface if the current process is the last one to use the uplink.

QNetworkRequest class

This class holds a request to be sent *QNetworkAccessManager*. It is part of the Network Access API and is the class holding the information to send a request over the network. It contains a URL and some ancillary information that can be used to modify the request.

QNetworkReply class

This class contains the data and headers for a request sent with *QNetworkAccessManager*. It also contains the data and meta-data related to a request posted with *QNetworkAccessManager*. Like *QNetworkRequest*, it contains a URL and header (both in parsed and raw form). *QNetworkRequest* is also a sequential-access QIODevice, which means that once data is read from the object, it is no longer kept by the device. When more data are received from the network and processed, the *readyRead()* signal is emitted.

QVariant class

This class acts like a union for most common Qt data types. Since C++ forbids unions from including types that have non-default constructors or destructors, most interesting Qt class cannot be used in unions. A *QVariant* object holds a single value of a single type at a time.

QVariant also supports the notion of null values. However, *QVariant* types can only be cast when they have had a value set. Since *QVariant* is part of *QtCore* library, it cannot

provide conversion function to data types defined in *QtGui* such as *QColor*, *QImage* and *QPixmap*.