

Oil IT Journal Contributed Paper

Title Soft Real-time Data Viewer using WITSML

Author William Sellick

Affiliation Aberdeen University

Publication Date July, 2008

DOI <https://doi.org/10.69894/900008>

ABSTRACT

This project is an investigation into the real-time elements of a data transfer standard called "WITSML". The project was initiated by a private company called IDS who required an investigation into this topic. IDS wished for selected technology to be used to evaluate their viability for future development.

RGU Honours Project

(Soft Real-time Data Viewer using WITSML)

William Sellick (0200389)

Supervisor: Deryck Brown

Declaration

I confirm that the work contained in this report has been composed solely by myself. All sources of information have been specifically acknowledged and verbatim extracts are distinguished by quotation marks.

Name: William Sellick

Date: 25/04/2008

Matriculation No: 0200389

Signature:

Abstract

This project is an investigation into the real-time elements of a data transfer standard called "WITSML". The project was initiated by a private company called IDS who required an investigation into this topic. IDS wished for selected technology to be used to evaluate their viability for future development.

The project involved the creation of a web-based system that collected and displayed real-time data. The system was divided into two main sections: the client and the server. The client dealt with the user interaction and displayed relevant content on screen. The server held all the business logic that would supply the client with data content and acted similar to a proxy-server cache and forwarding data.

The system was designed by detailing the components required and how each technology would interact. The design was split up into the two main sections described earlier and involved a detailed analysis of the requirements. A number of problems occurred when implementing the design due to limitations of the technology selected. Where possible the problems were solved but in a few cases no solution could be found. These cases were documented in the implementation section.

The project was evaluated on its usability and efficiency. The client user interface was analysed to compare its functionality with the guideline provided by IDS. An attempt was made to measure the efficiency of the server by profiling the memory and CPU usage while applying a high load. However problems occurred when using the profiling tool which meant the results could not be accurately obtained.

Overall the investigation obtained valuable details about the real-time elements of WITSML and showed that the selected technology could be used to develop an application to support it.

Word count: 22,996

Table of Contents

Abstract.....	3
Introduction.....	1
1.1 Sponsorship of Project.....	1
1.2 Aims of Project.....	1
1.3 Structure of Report.....	1
Background Information.....	3
1.4 Problem Area.....	3
1.5 Introduction to Web Services.....	4
1.6 Introduction to WITSML.....	5
1.6.1 Why WITSML is required.....	5
1.6.2 The structure of WITSML.....	6
1.6.3 WITSML and SOAP.....	9
1.6.4 The WITSML query language.....	10
1.6.5 WITSML real-time elements.....	11
1.7 Introduction to the Subscriber/Publisher Interface.....	11
1.7.1 Where and why the interface is used.....	12
1.7.2 Usage in this project.....	12
1.8 Introduction to Rich Internet Application.....	13
1.8.1 RIA Languages.....	13
1.8.2 Adobe Flex.....	13
1.8.3 Usage in this project.....	15
1.9 JBoss Application Server.....	15
1.10 JAXB.....	17
Requirements Analysis.....	18
1.11 Server Requirements.....	18
1.11.1 RIA Interaction Requirements.....	18

1.11.2 WITSML Client Requirements.....	19
1.12 Client Requirements.....	19
1.12.1 User Interface Requirements.....	19
1.12.2 The Subscriber/Publisher interface requirements.....	20
System Breakdown.....	21
1.13 System Overview.....	21
1.13.1 JBoss Application Server.....	22
1.13.2 Flex Data Services.....	27
1.13.3 Adobe Flex Client.....	28
Design Discussion.....	32
1.14 Server Design.....	32
1.14.1 RIA Interaction Design.....	32
XML Structure.....	34
System Core Extendibility.....	38
1.14.2 WITSML Client Design.....	40
WITSML Query Interface.....	41
WITSML Query Generation.....	41
WITSML Query Types.....	42
SOAP Client Configuration.....	43
SOAP Client Job Scheduling.....	43
Persistence Management.....	45
1.15 Client Design.....	47
1.15.1 User Interface Design.....	47
Sectioning Data into Different Forms.....	48
1.15.2 The Subscriber/Publisher Interface Design.....	51
Implementation Issues.....	52
1.16 Server Implementation.....	52
1.16.1 RIA Interaction Implementation	52

1.16.2 WITSML Client Implementation.....	53
1.17 Client Implementation.....	60
1.17.1 User Interface Implementation.....	60
1.17.2 The Subscriber/Publisher Interface Implementation	65
Evaluation.....	68
1.18 Client Evaluation.....	69
1.18.1 Login.....	69
1.18.2 Post-login.....	70
1.18.3 Main Display.....	72
1.18.4 Graph Display.....	78
1.19 Server Evaluation.....	82
1.19.1 Performance Profiling.....	82
1.20 System Evaluation.....	85
1.20.1 Technology Evaluation.....	85
1.20.2 Sponsoring Company's View	87
Conclusions.....	89
1.21 Analysis of Aims.....	89
1.22 Reflection on Achievement.....	90
1.23 Future Work.....	90
References.....	1

Introduction

This section introduces the project and its aims. Section 1.1 details the sponsorship of the project and how the design decisions will be influenced by the sponsoring company. Section 1.2 introduces the aims of the project and Section 1.3 concludes this section by describing the structure of the report.

1.1 Sponsorship of Project

This project will be sponsored and overseen by a private company called Independent Data Services (IDS). IDS are a small software company who develop reporting applications for oil companies. IDS wish for this project to be an investigation into technologies that they hope to use in future projects. Specifically, they require an investigation into Adobe's rich internet application language called "Flex" and the newest version of Redhat's application server called 'JBoss'.

1.2 Aims of Project

The main aim of this project is to investigate the real-time data elements that are available in the data transfer standard WITSML. The investigation should include a analysis of the WITSML standard and how modern technology can be used to develop an application capable of real-time data streaming.

A secondary aim of this project is to investigate technologies selected by IDS and evaluate their potential for usage in this style of application.

1.3 Structure of Report

This report consists of seven sections that relate to the various stages of development. Section 1.3 (Background Information) introduces the technologies required to understand the components used in the system. Section 1.10 (Requirements Analysis) analyses the requirements given from IDS to create a guideline of functionality required. Section 1.12.2 (System Breakdown) describes a breakdown of all the components required to achieve the functionality defined in Section 1.10. Section 1.13.3 (Design Discussion) analyses the components required

and discuss the design required to implement them. Section 1.15.2 (Implementation Issues) describes the issues faced when implementing the design previously described. Section 1.17.2 (Evaluation) evaluates the work achieved in the implementation and compares it with the requirements given from IDS. Section 1.20.2 (Conclusions) concludes the report by analysing the aims and discussing the achievement of the project.

Background Information

This section introduces the elements required to fully understand the development of components for this application. Section 1.4 introduces the problem area and describes the problem this project aims to investigate. The sections that follow section 1.4 give a brief overview of the technologies and standards used in the project.

Section 1.5 gives a brief description of web services and the protocols that are used. Section 1.6 introduces the WITSML standard and explains its usage and application in the system. Section 1.7 describes the subscriber/publisher interface, its usage in applications and where it will be used in this system. Section 1.8 introduces Rich Internet Environments and examines the selected technology for this system. Section 1.9 describes the JBoss application server, its important components for this project and how it will be used in the system. Section 1.10 briefly describes the XML handling technology JAXB and how it will be used in the system.

1.4 Problem Area

The oil industry is a diverse sector that consists of a variety of different areas of interest. This project is concerned with offshore platforms and the data they produce. Offshore platforms, or drilling rigs, are large structures usually based at sea that are used to extract oil and natural gas through wells on the sea bed. These structures house both the machinery required to extract the oil and the work force required to operate them.

When a company begins a drilling operation in a well, it must monitor the drilling equipment to improve the performance and ensure nothing dangerous happens. This is achieved by using sensors attached to each tool to measure and alter its current settings. Each of these sensors sends its readings to a centralised data centre, and they are recorded in a company-specific format.

The problem with this setup is that although a single company is sponsoring the drilling operation, they employ a wide range of contractors to perform specialist tasks.

Each of these contractors needs to exchange data with both the sponsoring company and other contractors to work effectively. However, each contractor will store the data slightly differently and will expect it in a different format. This can present significant problems in the coordination of the drilling operation, and requires a shared data representation to permit its exchange.

This project will investigate a shared data representation language known as WITSML. The intent of the project will be to develop a web-based application that uses this language to view data available on many different company's servers. In order to successfully retrieve data from these servers, web services will be used.

1.5 Introduction to Web Services

Web services are published API's that allow software systems to interact over a network. They are based on a standard created by the World Wide Web Consortium (W3C) that defines how the client and server communicate (W3C, 2004). The W3C are an organisation that develops specifications and guidelines to standardise technologies and their interaction between companies. They have made web services extensible by creating a dedicated language for developers to create their own services. This language is based on XML and validated against a schema created by the W3C. Service developers can create a WSDL (**Web Services Description Language**) file that defines the method names and parameters required for clients to communicate with their servers. Application developers can then use this file to generate program code that represents the definition in the WSDL. This code contains a basic implementation of a client or server but does not contain any business logic. This creates a framework that they can build from and develop their own custom applications.

The most common form of communication for web services is the use of XML that adheres to the SOAP standard. SOAP is a protocol used to exchange XML messages over a network and provides a foundation for web services to interact. SOAP typically uses HTTP for transport between servers. This works well within networks that are heavily firewalled as communication is increasingly difficult and using the same protocol as a standard web browser increases the chance of success. SOAP uses XML

because it is a wide spread technology that has been integrated into many systems. The extensibility of XML also allows standards to be created with an open specification, which is ideal for creating a non-profit standard.

The web services used in this project will be defined by the WITSML standard that this project will investigate. The WITSML standard is described in the following section.

1.6 Introduction to WITSML

WITSML (**Wellsite Information Transfer Standard Mark-up Language**) is the standard developed to solve the problem of a shared data representation between companies. WITSML is an XML-based language that allows industrial data to be represented in a predefined form and exchanged between companies. It was originally aimed at a large group of organisations (including oil companies, service companies, drilling contractors, application vendors and regulatory agencies) to allow the flow of technical information between companies. WITSML-enabled applications are becoming popular for a large number of organisations, and are almost a necessity for reporting-based applications.

A WITSML document represents a set of data that can be encapsulated within a single XML document. The document is used to represent a subset of the well-site domain. Static and real-time data can be represented within these documents. Static data is represented in an object-oriented fashion that embeds related data inside the same document. In contrast to this, real-time data is structured to allow quick additions to the document. These documents are usually split up into a header and data section. When new real-time data is collected, it is published by the server by adding a new value to the data section and updating the header to inform clients that new data is available.

1.6.1 Why WITSML is required

The volume of information used in the oil industry has grown at a steady rate as technology has become cheaper and more available. Historically the interchange of well-site data was performed by the serial transfer of ASCII data. This data stream was known as WITS (**Wellsite Information Transfer Standard**) and was the most

basic form of communication between companies. The documents produced for this stream were not user friendly and were difficult for humans to read. WITS did, however, allow basic data to be shared between the necessary parties.

As the volume of data increased it was clear that the ASCII stream was not sufficient to support the demand. A company, called Energistics, took up the challenge to modernise the WITS language and develop an XML-based standard for well-site data exchange. WITSML was the result of their work. WITSML is based on existing Internet standards (W3C, SOAP, WSDL, and XML) and is similar to other XML standards. The language is defined using XML Schemas that specify everything from the data objects that can be represented, to the units for each measurement. Using schemas to structure this language allows quick and accurate validation on the documents produced, and ensures that documents exchanged are in the correct format.

1.6.2 The structure of WITSML

A WITSML document represents groups of data about a subset of the well-site domain. It is a text-based XML document that conforms to the WITSML schemas. The document is structured in many layers that group elements together allowing humans to see the relationship between data. The top layer is defined in the schema documentation as:

"The WITSML API mandated plural root element which allows multiple singular objects to be sent. The plural name is formed by adding an "s" to the singular name" (Energistics, 2003)

This documentation expresses a general rule that requires all documents to contain a wrapping tag that includes all other elements. Code Fragment 2.1 shows two examples of the use of this tag rule. It shows a "wells" tag that contains a list of two well elements and an opsReports tag that contains a singular operation report. It is apparent from this example that the root element is simply the containing element's name suffixed with an 's'.

```
(Document 1)
<wells ...>
  <well ...>
</wells>
```

```

        <well ...>
        </well>
</wells>

(Document 2)
<opsReports ...>
    <opsReport ...>
    </opsReport>
</opsReports>

```

Code Fragment 2.1 – An example of the typical structure of a WITSML document.

The root element's tag is used to group all other elements together and allows multiple child tags (which represent the same singular well-site subject) to be included. The tag can contain an unlimited list of children but these must be of the same type. It must also contain a version attribute that defines which version of WITSML the document is using.

The next layer in the document encapsulates a specific aspect of the well-site in a tag; these are the `well` and `opsReport` tags in Code Fragment 2.1. This tag can contain multiple child elements that can either be of a complex or simple nature. The simple child elements are referred to as fields. Each field is defined as a specific data type that can vary from a string to a value from an enumerated list. Fields can also contain attributes that add extra information about the field (for example the unit in which the field is expressed). The complex child elements are grouping tags that share a similar structure to their parent tag and represent a more specific subject related to its parent's subject. These elements can be referred to as the third layer in the document as they can contain fields, further complex child elements and attributes.

A document's complex elements can be identified by a `uid` attribute. This value must be unique for each object and is mandatory when generating a WITSML document.

An example of a WITSML document is shown below in Code Fragment 2.2.

```

<wells xsi:schemaLocation="http://www.witsml.org/schemas/131
../obj_well.xsd" version="1.3.1.1">
    <well uid="w-12">
        <name>6507/7-A-42</name>
        <country>US</country>
        <timeZone>-06:00</timeZone>
        <operator>Operating Company</operator>
        <statusWell>drilling</statusWell>
        <purposeWell>exploration</purposeWell>
        <dTimSpud>2001-05-31T08:15:00.000</dTimSpud>
        <wellDatum uid="SL" defaultElevation="true">
            <name>Sea Level</name>
        </wellDatum>
    </well>
</wells>

```

```

        <code>SL</code>
    </wellDatum>
    <wellLocation uid="loc-1">
        <wellCRS uidRef="proj1">ED50 / UTM Zone 31N</wellCRS>
        <easting uom="m">425353.84</easting>
        <northing uom="m">6623785.69</northing>
        <description>Location of well surface point in projected
            system. </description>
    </wellLocation>
</well>
</wells>

```

Code Fragment 2.2 – An example WITSML document. (Energistics, 2003)

The example shows a WITSML document which represents a 'well' (a physical location on the cartographic map). The top-level element is a grouping tag that encapsulates the singular `well` object and could allow further `well` objects to be added. Inside the `well` object there are many elements: some are simple types while the others are more complex. Simple types are expressed with a single tag and encapsulate a single property of the parent object. Code Fragment 2.3 shows an example of two simple elements. The `name` tag gives a string value of the `name` assigned to the well. The `dTimSpud` tag gives a date value of when work on the well began.

```

<name>6507/7-A-42</name>
<dTimSpud>2001-05-31T08:15:00.000</dTimSpud>

```

Code Fragment 2.3 – Simple elements from a WITSML document.

Complex types are expressed by tags that contain other tags within them. These tags represent a more specific subject of parent's topic. Code Fragment 2.4 shows an example of a complex element. The `wellLocation` tag is a sub-topic of the `well` object and contains details about the location of the well.

```

<wellLocation uid="loc-1">
  <wellCRS uidRef="proj1">ED50 / UTM Zone 31N</wellCRS>
  <easting uom="m">425353.84</easting>
  <northing uom="m">6623785.69</northing>
  <description>Location of well surface point in projected system
  </description>
</wellLocation>

```

Code Fragment 2.4 – Complex elements from a WITSML document.

Each field gives a single piece of information about the `well` object. The complex child elements (`wellDatum` and `wellLocation`) contain groupings of fields or more specific subjects of the `well` object. Each child element contains fields that give further information about the specific subject.

1.6.3 WITSML and SOAP

WITSML was conceptually designed to use SOAP as its main protocol for transmitting documents between systems. The developers wanted to use a modern protocol that had scope for future improvement so that WITSML could evolve in the future. The WITSML standard includes an associated WSDL file. This WSDL defines the interfaces available for both the client and server implementations of the standard.

For the server, the WSDL defines the methods it should publish and the parameters that should be accepted. The server can publish the following methods:

- *WMLS_AddToStore* - This adds values into a store
- *WMLS_DeleteFromStore* - This deletes values from a store
- *WMLS_GetBaseMsg* - This returns a message from an error code
- *WMLS_GetFromStore* - This retrieves values from a store
- *WMLS_GetVersion* - This retrieves the version of the server
- *WMLS_UpdateInStore* - This updates existing data in a store
- *WMLS_GetCap* - This describes the capabilities of a server. It lists

everything the server can do as well as which objects are supported.

For the client, the WSDL defines what methods can be invoked and what parameters to send. The client uses the methods stated above to query for information from the server.

1.6.4 The WITSML query language

The method used by clients to query a WITSML server is defined in the WITSML API document. Clients must supply a valid query that the server can then interpret and return the requested data. The WITSML query structure is exactly the same as the document structure but is interpreted differently. Queries are interpreted in the following manner:

1. If an element is populated in a document this means that it is part of the selection criteria used to choose the relevant data.
2. If an element is included but has no value the server should return this element with its value populated.
3. If an element is not included it should be ignored.

Once the server receives a valid query, it populates the document and returns it to the client.

Code Fragment 2.5 shows an example query that could be used to retrieve the data shown in Code Fragment 2.2. The query contains a populated UID attribute and name element. These fields will be used as selection criteria on the WITSML server being queried. The other simple fields are not populated and so will be treated as required fields that the server should populate. The query also contains complex elements that are not populated. These will be interpreted as further required fields and populated by the server. One of the complex objects also contains an unpopulated simple element; these are treated in the same way as the other simple elements. Some of these simple elements contain an attribute called `uom`. This attribute tells the server which unit the client expects the value to be expressed in. This type of attribute must appear in all simple elements that contained measured values.

```
<wells version="1.3.1.1">
```

```

    <well uid="w-12">
      <name>Well 1</name>
      <country/>
      <timeZone/>
      <operator/>
      <statusWell/>
      <purposeWell/>
      <dTimSpud/>
      <wellDatum uid="">
        <name/>
        <code/>
      </wellDatum>
      <wellLocation uid="">
        <wellCRS/>
        <easting uom="m"/>
        <northing uom="m"/>
        <description/>
      </wellLocation>
    </well>
  </wells>

```

Code Fragment 2.5 – An example WITSML query

1.6.5 WITSML real-time elements

WITSML can be used to stream data from continually updating sources. These sources (also known as data aggregators) collect information from multiple sensors and amalgamate it into a single storage system. A WITSML client can be used to continually query these aggregators to receive the latest updates.

WITSML clients query the data aggregators for WITSML objects that support real-time streaming. These objects are structured in an expandable manner so that they can be updated easily. This is important as the produced volume of data can be quite high and the server must be able to update the object efficiently. As well as being structured with dynamic updates in mind, these WITSML objects also contain key fields that are used to check if new data has been published. If these fields change it is a sign to the clients that new data is available and they need to query for the next update.

1.7 Introduction to the Subscriber/Publisher Interface

The subscriber/publisher interface defines a system setup that allows multiple applications to connect to a feed of information. This type of connection allows clients to subscribe to topics of data and whenever data is published, the server transmits it to all the subscribed clients. The interface contains two key components: a publisher and subscriber.

- A publisher is a component that extracts data from an input source and publishes it for others to view. In the oil industry, an input source would typically be a sensor attached to equipment that would measure one or more of its attributes. Once the publisher has obtained the value, it inserts it into a temporary storage medium known as a topic. The server then forwards the new value to clients that are subscribed to that topic.
- A subscriber is a component that can subscribe to multiple topics to receive data whenever it is published. A client is only required to subscribe to a topic and does not have to continually query the server for new information. It is the responsibility of server to push new information to clients.

1.7.1 Where and why the interface is used

This interface is used when data needs to be streamed from a source to a single or multiple destinations. It would typically be used when multiple clients require access to the same data stream. A basic client configuration would need to continually query a data source for new information. By using this interface it removes the complexity from the client and allows it to simply wait for updates. This allows clients to be created that can view a large number of streams with only a small overhead for each client.

The main reason this improves the performance of the subscribed client is because the program code required to process the data is based on the server. This delegates a lot of responsibility from the client and allows it process a higher volume of data. It is important to note that although this interface scales well for small systems, once the amount of topics reaches a medium value it starts to strain the server's resources. Therefore, this interface is not suitable for large scale projects.

1.7.2 Usage in this project

This project will use this interface between the application server and the client. The project will contain elements that will require real-time data streaming. These streams will occur at different time intervals and will be the direct feeds for rendering graphs. As multiple clients will be able to view the same data streams, it is important to reuse

resources that are available already. Using topics and the subscriber/publisher interface allows streams of data to be shared throughout the project, reusing the resources available.

1.8 Introduction to Rich Internet Application

Rich Internet Applications (RIA) are web-based applications that provide the functionality of desktop applications. The term “Rich” comes from a direct comparison of the simple web-based pages that did not provide much functionality. In comparison these pages provide a large set of functions to enable the user to do more. These applications usually transfer the minimum information necessary for the user to interact with the program and maintain the majority of the data (the state) back on the server. RIA’s usually run in a web browser and do not require any installation files except plug-ins based in the browser.

1.8.1 RIA Languages

Since the launch of Web 2.0 RIA’s are steadily becoming more popular. Many languages have emerged since this point to fulfil the need for better and more complex user-interfaces. As RIA’s usually run in a web browser the language they are written in is usually based on a technology that will compile into a format a web browser can understand. The two most popular formats are DHTML and Flash animation. DHTML is an extension of static HTML content that allows more complex features to be integrated into web-pages. Flash based languages allow developers to create object-oriented programs that compile into a Flash animation. This can be embedded inside a web-page as a Flash movie.

As IDS has requested an investigation into Adobe Flex, this product will be used to create the RIA for this project. The features and components of this product will be examined further to give an idea of its capabilities.

1.8.2 Adobe Flex

Adobe Systems has recently produced a powerful framework called Flex 2 for creating RIAs (Borck, 2006). The application used for developing programs in this language is based on the Java-based Eclipse, an Integrated Development Environment (IDE). This

is an important aspect for the project as the server will be based in Java and will allow elements to be shared between the development of the server and the client. The builder uses MXML (Macromedia Flex Markup Language) code to build up elements and merges this with ActionScript to gain executable code behind the user interface. Actionscript is an Adobe Flash based language similar to other scripting languages like Javascript or VBScript.

The Flex 2 framework provides a versatile user interface for web development. It provides the following features:

- *Typical UI components* – A selection of graphically represented components which can be dragged and dropped on a canvas.
- *Data Grids* – A way to represent tables of data in an ordered and tangible manner.
- *CSS Support* – The view of pages can be controlled in a similar manner to HTML allowing mark-up and customisation of components.
- *XML Support* – Adobe Flex uses the ECMAScript for XML (E4X) extension to parse XML and generate Flex objects to represent XML elements. E4X is a standardised extension used in many scripting languages to handle XML processing and provide access to XML elements in a fast and efficient manner. These Flex objects contain methods that allow the content of the XML to be referenced programmatically.
- *Web Application Support* – Provides the ability to use web services without the need for server-side code. The Flex language includes core libraries to enable Flex applications with web-based communication. The HTTP communication implementation is called "HTTPService". This object contains the features required to connect to other web-based applications. It encapsulates the ability to invoke a call on a selected URL and maintains the connection between the two applications.
- *Subscriber/Publisher Support* – The product includes a deployable Java web project that can interact with compiled Flex files. This project is known as the Adobe LiveCycle Data Services (LCDS) and can be deployed on any Java web container. It includes features for web services management. One of these

features is the facility for subscription to subscriber/publisher based technology. The current version of this project supports the Java Messaging Service (JMS) and Action-script messaging for subscriber/publisher interaction.

- *Chart Support* – The product can include an additional module called Flex Charting. This product includes a collection of chart implementations suitable for use in Flex applications. These charts can be easily styled through the use of Cascading Style Sheet (CSS) parameters. Charts typically consist of two axes and a collection of data series. A data series is represented by a collections object (Lists or Arrays) that contains a list of data about a specific topic usually ordered by an attribute of the data. Multiple data series can be added to each graph to allow comparison between the data.

1.8.3 Usage in this project

This project will use an RIA to display client information. It will harness the additional functions of an RIA to extend the user-friendliness and make the overall design easier to use. It will also use the more advanced features of an RIA to allow the subscriber/publisher interface to be integrated into the program.

1.9 JBoss Application Server

The JBoss application server is an implementation of the Java enterprise edition specification. It can be described as a complex web-server that provides a number of features to Java-based application. The application server uses Java objects known as Session Beans to maintain efficient processing and load balancing of the server. Session beans are Java objects that contain extra information in order to tell the server how to process them. In the latest version of JBoss this information is in the form of Java 5 annotations. Annotations are extra elements defined by an '@' symbol that are found inside a Java file to give further information about the class or an attribute of the class. The application server provides a naming service in order to reference session beans. This service allows objects to be bound to unique names and be looked up in any class inside the server.

The application server can support a number of different modules that provide various features when utilised. A number of these modules and their features are listed below.

- *Web Container Handling* – The JBoss application server provides a module to publish Java content to a specified URL and port number. It requires a dynamic web-project to be added to the application with the Java content contained within it. This content is then published using a module known as the web container. In order to publish content, the Java classes must be in a set format defined by the servlet specification written by Sun Microsystems.
- *Messaging Service* – The application server contains a module known as JBossMQ that is an implementation of the Java Message Service (JMS) API. This component can be used to create, update and destroy Subscriber / Publisher topics of data and then publish them on the server so that they are available for client interaction (OutwardMotion, 2007).
- *Web Services* – The JBoss application server allows the integration of a web services module called Apache Axis 2. Axis 2 is an implementation of the SOAP protocol defined by the W3C and contains libraries to enable the application to communicate through web services. It also contains automated processes to generate Java implementations of both a client and a server from a WSDL file. However, the code produced only contains a basic implementation of the objects required to communicate with a server.
- *Job Scheduling* – The application server includes a module called Quartz for scheduling jobs. Quartz is an implementation of a job scheduler that allows jobs to be executed at set intervals or at set times.
- *Java Persistence* – The application server contains a module known as Java Persistence that handles the configuration and interaction of databases within a Java application server (RedHat, Unknown). It uses Enterprise JavaBean (EJB3) to map database tables and represent them with Java objects known as Entity Beans. In order to create entity beans, Java 5 annotations are used to add additional information to the classes. These annotations are scanned by the

module at the start-up of the server to build up a library of supported tables/objects.

1.10JAXB

Java Architecture for XML Binding (JAXB) is one of the most popular Java libraries used to process XML documents. It uses XML schemas to generate Java classes that represent the tags used in an XML document (Ort, et al., 2003). It allows data to be automatically transformed between XML and Java and provides the tools required for generating Java classes from an XML schema. This transformation process is achieved by adding extra data into the Java classes generated. The latest version of JAXB uses Java 5 annotations to provide this data.

Requirements Analysis

As this project is an investigation into specific technologies, the system produced could include a vast number of features. However, to limit the scope of the project and to conform with the needs of the sponsoring company the requirements will be analysed to produce a requirements analysis. This section is broken into two sections: Server Requirements and Client Requirements. The server requirements describe the features that the server must fulfil while the client requirements specify the functionality that is required by the client.

1.11 Server Requirements

The server requirements consist of two main parts: the RIA Interaction and the WITSML client. The RIA interaction section encapsulates the elements of the server that need to generate and maintain content for the client. The WITSML client section includes the elements of the server that are required so that WITSML data from other servers can be extracted, processed and saved. The server must bridge these sections to allow data to be passed from the WITSML client to the RIA.

1.11.1 RIA Interaction Requirements

1. Provide a web-enabled access point for a RIA to connect to.
2. Exchange information with a RIA client using a defined XML structure.
 - 2.1. The XML must be able to represent full and partial screen data so that the RIA can render a complete display.
 - 2.2. The XML must be able to represent all labels used onscreen and structure them in a way the RIA can efficiently process.
 - 2.3. The XML must be able to represent lists of items that can be used in dropdown menus and selection boxes.
3. Provide an interface for real-time data to be streamed to an RIA.
 - 3.1. Provide the ability to publish multiple topics of real-time data
 - 3.2. Provide the ability to have multiple connections to the same published topic.
4. Allow easy maintenance and future extendibility to the core business logic.

1.11.2 WITSML Client Requirements

1. Provide an interface to generate and execute a WITSML query.
2. Generate WITSML queries for all required WITSML objects.
 - 2.1. Generate queries with parameters specified by the RIA.
 - 2.2. Generate 'complete' queries of an entire well-site subject.
3. Provide an implementation of a WITSML SOAP client for retrieving data.
 - 3.1. Retrieve logs and trajectories from other WITSML servers.
 - 3.2. Define XML based business logic to link WITSML subjects together.
 - 3.3. Provide job scheduling to query for updates on selected WITSML objects.
4. Convert data from a WITSML format into the required database model.
 - 4.1. Save the collected data as it is streamed to the WITSML client.
 - 4.2. Reload data that has been saved by using criteria supplied by the RIA client.

1.12 Client Requirements

The client requirements consist of two main parts: user interface and the Subscriber/Publisher interface. The user interface consists of all the other elements required to display and update data. The subscriber/publisher interface includes the configuration of data services to maintain real-time streams between the client and the server.

1.12.1 User Interface Requirements

1. Provide the ability to connect to a web application.
2. Invoke queries for data by clicking on elements on screen.
3. Handle a predefined XML format.
 - 3.1. Extract labels from the XML to display titles, labels and buttons.
 - 3.2. Extract lists from the XML to populate combo boxes.
 - 3.3. Extract relevant sections to update all or part of the screen.
4. Provide a graphical representation of static and real-time data.
 - 4.1. Display a 2D graphical representation of drilling parameters.
 - 4.2. Display a pseudo 3D representation of the trajectory of a Wellbore.

- 4.3. Allow both axes to be defined by the XML data.
- 4.4. Update the graph in real-time.
- 5. Provide a tabular representation of static data.
- 6. Display a knowledge bubble which contains relevant information to the current selection.
- 7. Display a text window that can show related data to the current selection.

1.12.2 The Subscriber/Publisher interface requirements

- 1. Provide an endpoint for real-time data to be streamed.
 - 1.1. Allow the ability to subscribe and unsubscribe from streams of data.
 - 1.2. Clean up all connection related objects created to avoid memory leaks.

System Breakdown

This section contains a breakdown of the project to help explain how the requirements link to the technology selected. Section 1.13 gives a basic overview of the components required. Section 1.13 builds on the overview in Section 1.13 and expands it to explain how each component will be constructed.

1.13 System Overview

The system is divided into 4 main sections.

- WITSML Data Source – A WITSML data source is a data server that allows a web service to be used to interact with its published data. It will act as a source for all data (both static and real-time). This section will not be implemented for this project but is a requirement to feed the project data.
- JBoss Application Server – This section contains all the main logic required for achieving the server requirements. The server will source, process, save and then forward all data from WITSML servers to the Adobe Flex client. It will implement all the requirements for the WITSML client and RIA interaction.
- Flex Data Services – This section handles the interactive data services required by the Adobe Flex client. As the client is based inside the browser it cannot easily maintain a Subscriber/Publisher connection and therefore needs this application to manage the connection for it.
- Adobe Flex Client – This section handles the display of data inside the browser that the user interacts with. The client runs inside the Flash plug-in and allows advanced features to be used to display and interact with the data.

Detailed System View depicts how the various sections link together. It also encapsulates the main elements that will be included inside each section to fulfil the requirements stated. Each section will be broken down and discussed in the following section:

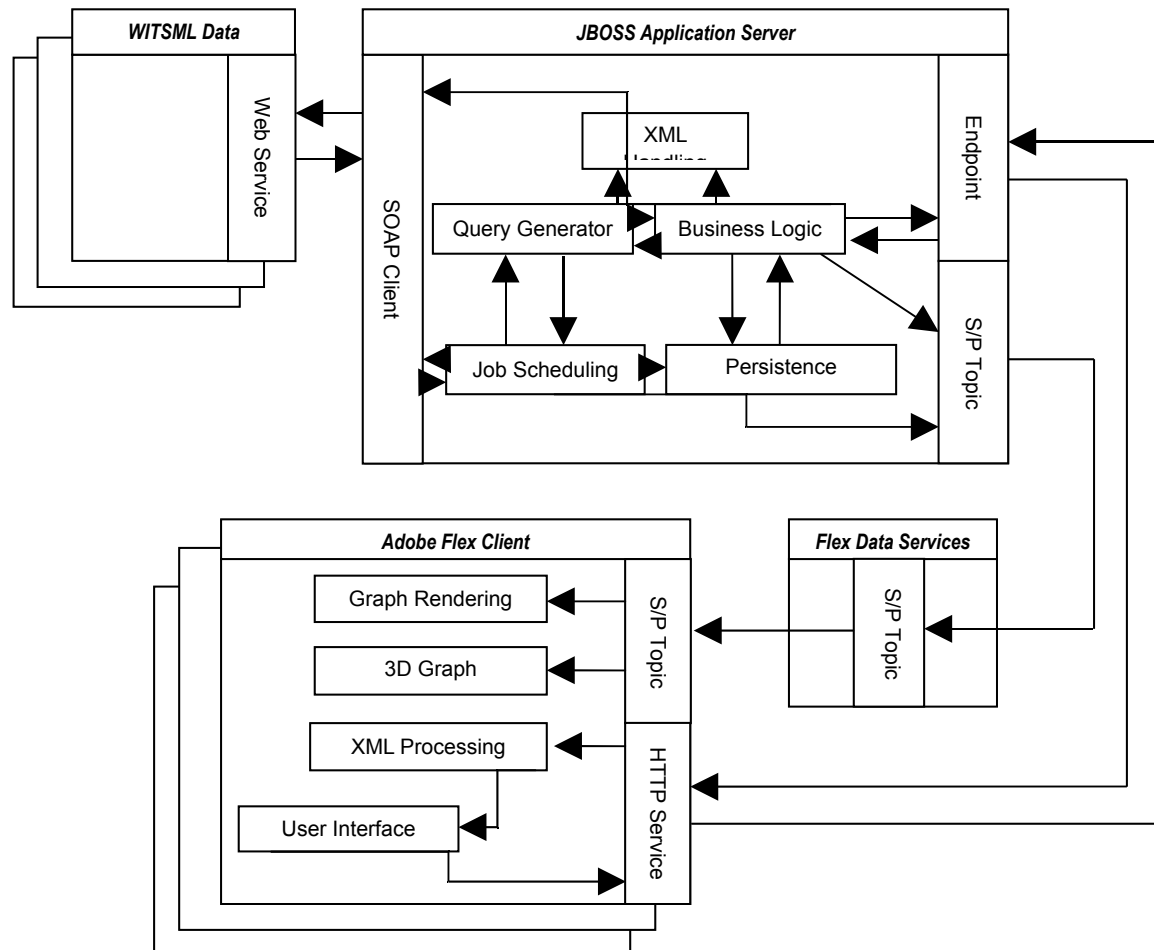


Figure 4.1 – A detailed system view.

1.13.1 JBoss Application Server

JBoss is a Java-based application server that provides a number of services. This system will use a number of these services as well as integrating other libraries to fulfil the server requirements. The server is made up of four key components.

1. **Business Logic Component** – This component is the controlling element inside the server and can be considered as the core of the server. It has direct access to all other components to be able to achieve any processing that the client may need. It is configured by an XML file that defines the functionality

available and the configuration for all other components.

Figure 4.2 shows how this component interacts with other components in the system. The business logic component is directly linked to all the key components of the system as it directly controls how they should interact with each other.

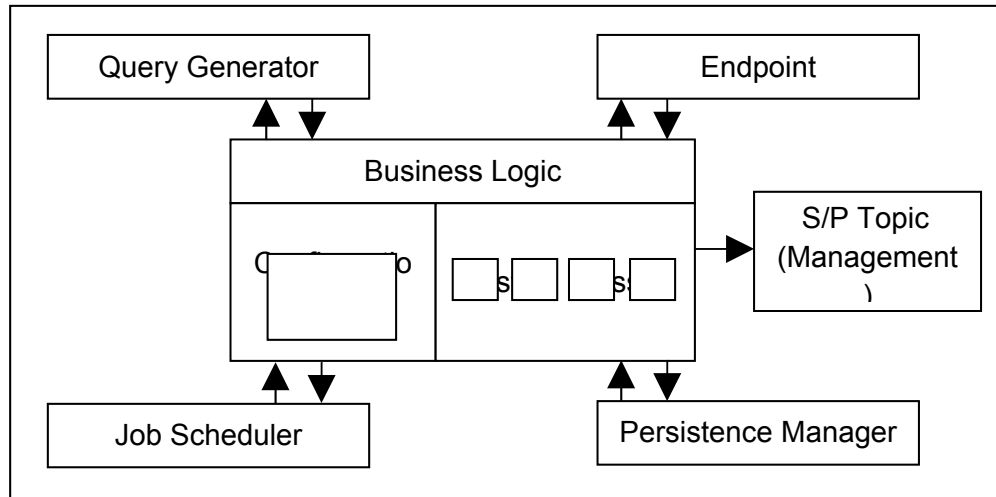


Figure 4.2 – A detailed diagram of the business logic component.

2. **Job Scheduler Component** – This component handles the management and execution of jobs in the server. It has access to the Query Generator, SOAP Client, Persistence Manager and JMS Topic components. The main purpose of this component is to continuously execute jobs to capture data as it changes or is updated on a data server. Each job executed by the scheduler shares these components to be able to create a Subscriber/Publisher producer. This producer supplies data to a Subscriber/Publisher Topic to feed data in real-time to subscribed Flex clients.

Figure 4.3 shows a breakdown of how this component uses the others to maintain up-to-date data. The job scheduler component manages job objects. Each job object has access to the SOAP client, query generator, persistence manager and S/P manager. This allows it to read data from a WITSML server,

cache the value and then add it to a S/P topic.

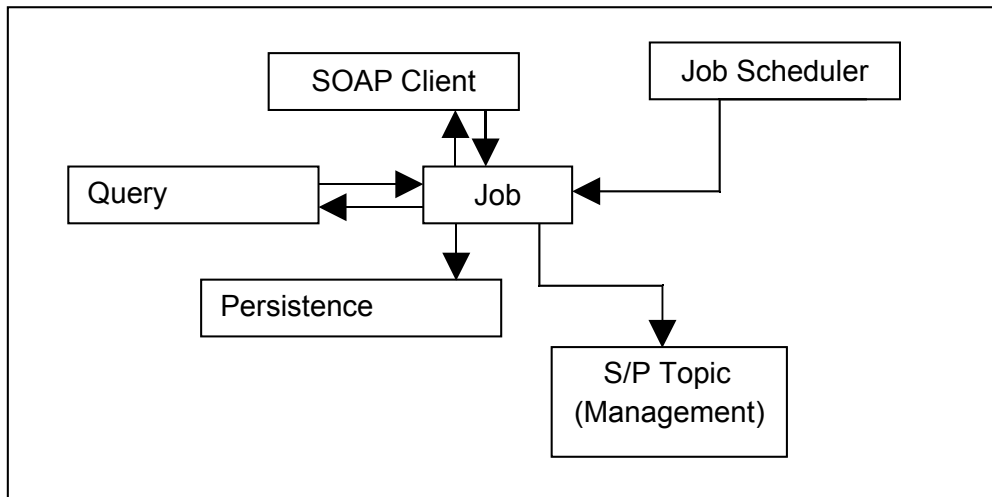


Figure 4.3 – A detailed diagram of the job scheduler component.

3. **Persistence Manager Component** – This component handles the loading and saving of data between the system and a database. It uses the built-in functionality of JBoss with custom configured classes to save and reload data into Java objects.

Figure 4.4 shows the persistence manager structure used in the server. The majority of this component will rely on the database management code supplied by the JBoss application server. It provides facilities to map custom written Java classes to database tables. The persistence manager controls this

configuration to provides functions to load, save, update and delete records.

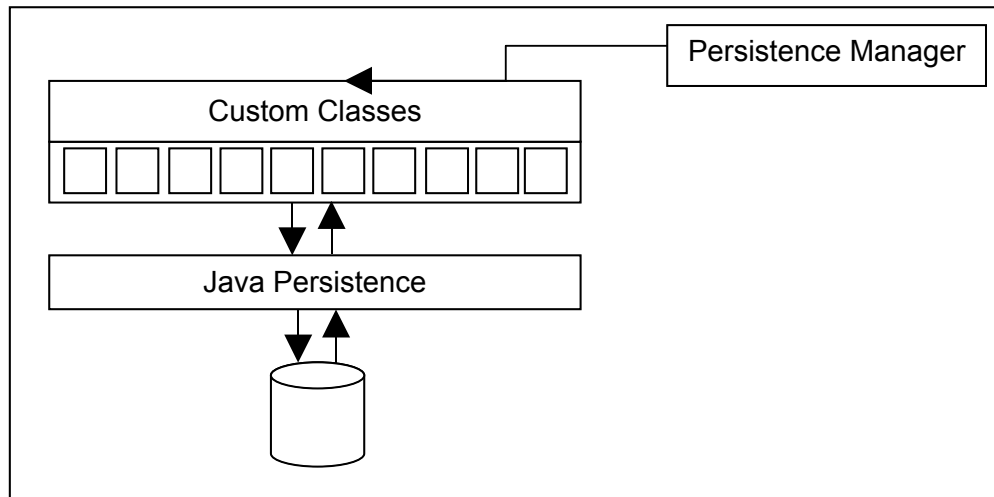


Figure 4.4 – A detailed diagram of the persistence component.

4. **XML Handling Component** – This component handles the processing of XML, converting it between the required forms. As the system represents the majority of elements in XML, this component is needed to load and convert XML in set formats.

Figure 4.5 gives a breakdown of how this component functions. The component is broken into two main parts: a marshaller and unmarshaller. The unmarshaller handles the conversion between XML documents and Java classes. The marshaller is the opposite of this and handles the conversion between the Java

objects and an XML document.

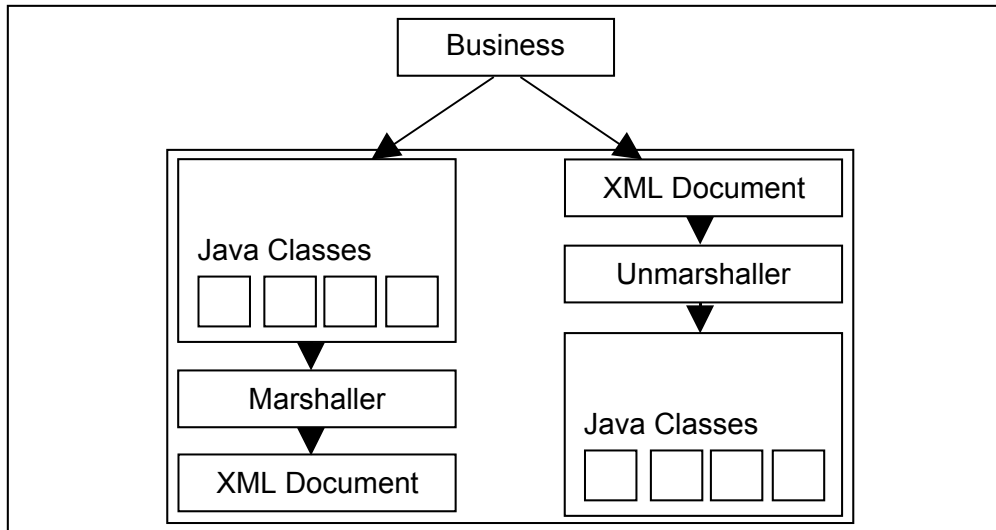


Figure 4.5 – A detailed diagram of the XML handling component.

1.13.2 Flex Data Services

Flex Data Services (also known as LCDS) are an extendable implementation of web services that have been provided for a Flash-based language. These services have been written in Java and are configured using XML. Although the API is documented online, the Java implementation is not open-source. This means that if this project requires the web services to be customised it could be difficult to extend the existing code.

The main purpose of this section is to maintain and clean-up connections that are used to stream data between the server and client. It is a key element as it provides a data forwarding service to the Flex client and removes the complexity from the Flash component.

1.13.3 Adobe Flex Client

Adobe Flex is a Flash-based language that can be compiled into Flash movies to produce an RIA. The language comes with many additional libraries to help develop and use components required for a rich user interface. The language also includes a chart library that allows charts to be developed and customised dependent on the user's requirements. This project will use a number of features available to create an RIA capable of fulfilling all of the client requirements. The client is made up of three main components.

1. **User Interface** – This component handles all of the displayable content that is available to the user. It maintains the correct view of data and hides/shows component as different events are triggered.

The user interface will consist of a number of elements. These elements have been defined by the sponsoring company to evaluate selected features of the Flex language. The following components are required:

- *Menu System* – A component is required to allow commands to be invoked from a standard menu like interface. It would be preferential if the menu was similar to the standard GUI menu that is available in most applications. This allows users of the system to become easily familiar with it as they will have used similar applications.
- *3D Visualisation of Wellbore* – A 3D component is required to be able to render three dimensional coordinates of the trajectory of a Wellbore on the GUI. It needs to be able to change the perception of the component in order to view the rendering from different angles. It will also need to be updated in real-time
- *Knowledge Bubble* – A component is required to expand on the details displayed in the 3D visualisation. This component will contain a table like structure with data values related to a selection in the 3D visualisation.

- *Text Window*– A component is required to give additional information about the details display in the 3D visualisation and knowledge bubble. Unlike the knowledge bubble this component will contain more abstract data that is not directly related to the 3D visualisation.
- *2D Representation of Drilling Parameters*– A 2D graph component is required to represent the status of drilling parameters while a wellbore is being drilled. This component needs to be updated in real-time by extracting two dimensional coordinates from data sent to it from the server. and Figure 4.7 depict a sample layout of the components required. Figure 4.6 shows a window with a menu at the top left corner. At the top of the window there are three selection boxes used to select which data the user requires to view. The main content of the screen is contained within a tabbed component, for this diagram the Wellbore tab is selected. The tab contains three of the components required: the 3D graph, the knowledge bubble and the text window. Figure 4.7 shows the same window as Figure 4.6 but with the Drilling Parameters tab selected. This tab contains three 2D graphs with additional components that allow the name of a stream of data to be selected and buttons to start and stop the streaming of data.

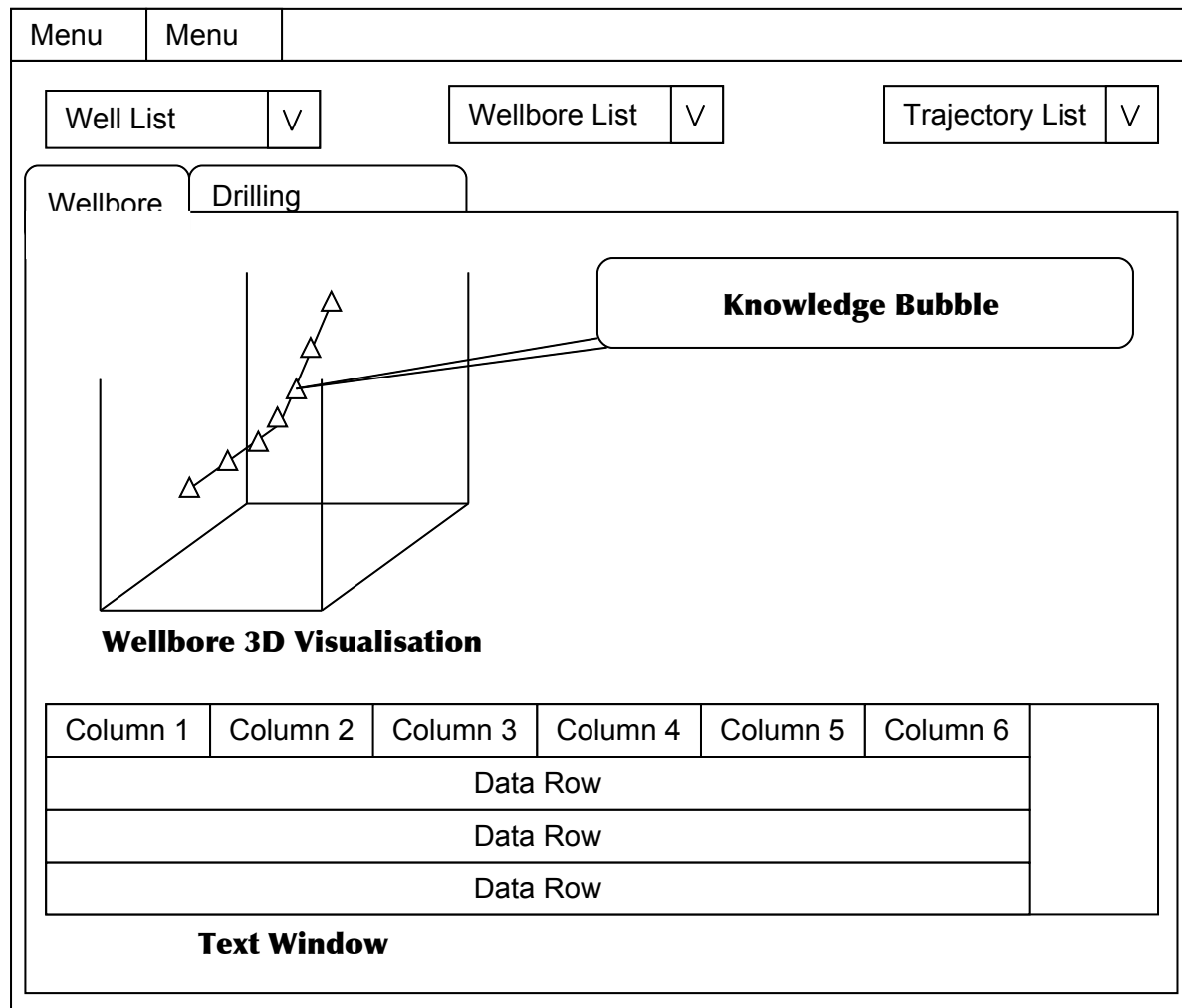


Figure 4.6 – An example layout of the design elements required by the sponsoring company.

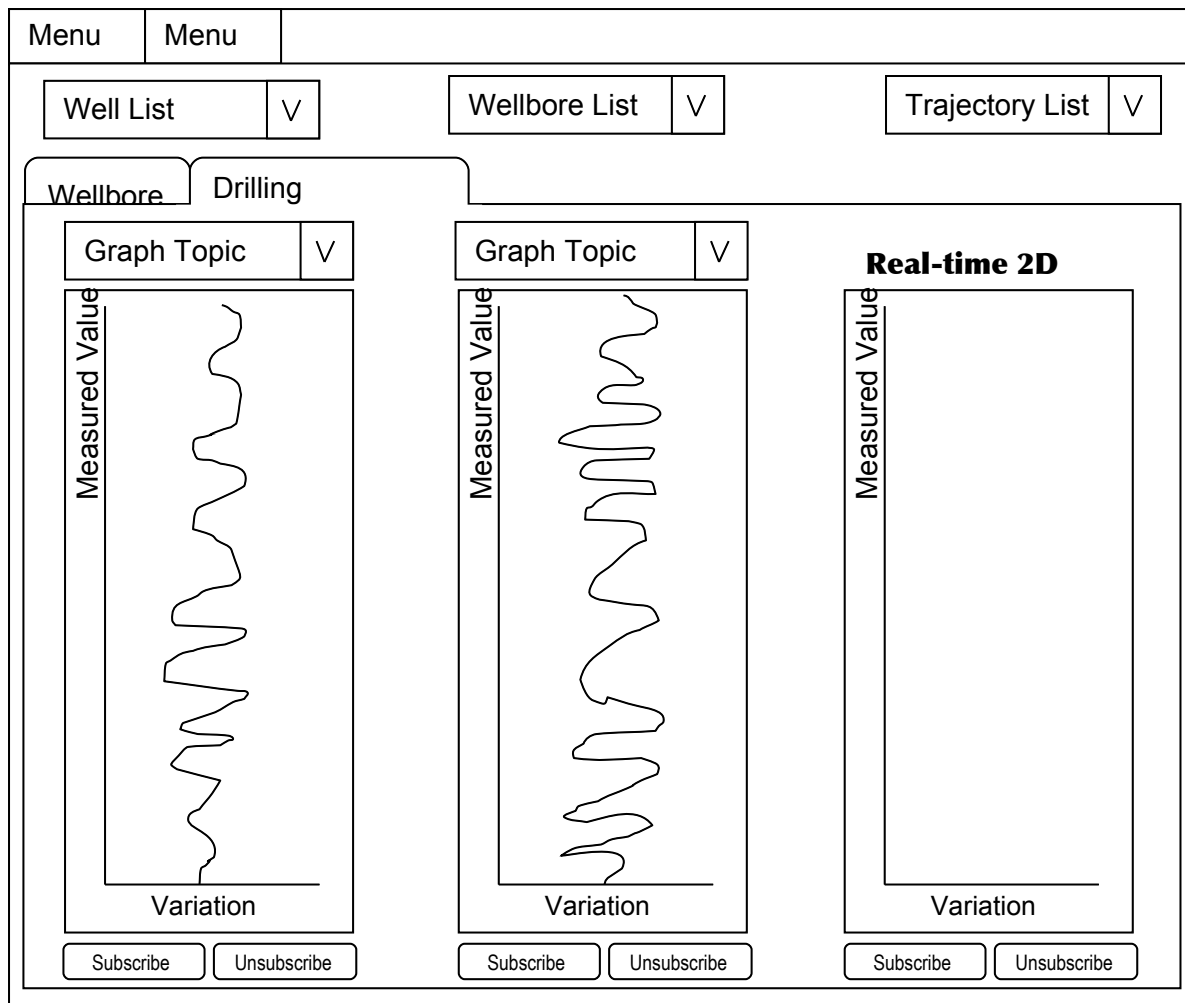


Figure 4.7 – An example layout of the real-time data graphs required.

2. **Graph Rendering** – This component handles the streaming and filtering of data passed from the Flex Data Services to the client. It maintains the data previously collected and processes any new data received. It is a requirement that the graphs are rendered in real-time therefore this component ensures the efficiency of the data processing and graph rendering.
3. **XML Processing** – This component handles the parsing and processing of content received from the server. All data sent from the server is defined in a set XML format. This component extracts data from the XML document and populates the relevant user interface items. It handles items like lookup lists, menu items, static graph data and the content for labels.

Design Discussion

This section contains a discussion of the design options available to implement the requirements specified in the Section 1.10. It will evaluate the implementation possibilities and decide on the best solution to fulfil the requirements.

1.14 Server Design

As the server will be based in JBoss the available features of the application server will be used to reduce the implementation of additional components.

1.14.1 RIA Interaction Design

5.1.1.1 Server Communication

Requirement 1.11.1 of the RIA Interaction requires a web-enabled access point to be created. In order to achieve this, three points must be satisfied; these points are described in the following list.

1. *Publish Java classes so they can be accessed by a URL.*

As this project will use a web-container to publish classes, the design needed for this requirement is minimal. It can be met by creating and deploying a custom servlet to interact with the RIA client.

2. *Create a generic interface so that the RIA can interact with the server.*

The JBoss naming service will be integrated into the business logic to give the RIA client information about which class to invoke when it needs different types of data. Using a Java interface, a standard method will be applied to all classes that could be invoked. This will allow a generic interface to be created inside the deployed servlet.

3. *Connect the published Java code to the business logic in the project.*

This requirement is heavily linked to the generic interface mentioned in the previous functional requirement. The business logic in the core of the project will contain a mapping of classes that the client can use to perform different

actions on the server. When a client first initialises it will need to receive the mapping so that it has the information to make further calls. When a client requires an action to be performed it will invoke a call to the server that contains the name and method of the class that will do the work. The servlet uses this information to lookup the class using the naming system and invokes a standard method to beginning the processing.

Requirement 3 of the RIA Interaction requires a real-time data interface to be created. A managing class that will wrap around the JBossMQ component will be implemented to fulfil this requirement. The managing class will provide the following features.

1. *Provide the ability to publish and remove topics.*

The server will need to create and destroy topics as they are required by the client. The managing class must provide methods to allow topics to be added and removed from business logic classes.

Topic Management

Topics must have unique names to identify them and allow them to be deployed on the server. Each topic will be published by using the unique identifier of the WITSML object that the real-time data stream is coming from. When multiple topics are linked to the same WITSML object but use a different field, the unique identifier will also contain the name of the field.

Memory Management

The server implementation must monitor the published streams and the number of clients connected to ensure that unused data streams are removed from the server. This is an important factor as potentially large quantities of data could be received and if this data is not being used by any clients, it is wasting the resources of the server.

2. *Provide the ability to have multiple connections to the same topic*

The JMS topic implementation allows multiple clients to subscribe to the

same topic. By using this technology this requirement will be satisfied without having to implement further code.

5.1.1.2 Exchanging XML data with the client

XML is a highly configurable language that allows a variety of types of data to be represented within it. This project requires data to be represented in a structured manner so that the client can easily extract it and populate elements on screen. The XML structure will be based around the data requirements of the client and be created through the implementation of an XML schema.

XML Structure

The XML structure must include the following elements.

1. *The ability to represent data for a specific section of each client view.*

The viewable content will be split up in regards to the screen it is displayed in. The top level element will group all the viewable content under a `views` tag. This tag will contain multiple `view` child tags, each one representing a single screen. A `view` tag can contain data about any possible element displayed on that screen. A `tab` tag will be added to the `view` tag to represent a grouping of elements and can contain multiple `element` tags. Each individual element on screen will be represented within an `element` tag that will contain the data required to render this element onscreen.

Code Fragment 5.6 gives an example of the XML structure described above. This XML fragment describes a single screen with two `tab` elements contained within it. Each `tab` also contains child elements which in turn contain data about those elements.

```

<views>
  <view id="mainView">
    <tab name="wellbore">
      <element id="wellbore-graph" type="graph">
        ...
      </element>
      <element id="traj-details" type="table">
        ...
      </element>
    </tab>
    <tab name="drill-params">
      <element id="drill-params-graph" type="graph">
        ...
      </element>
    </tab>
  </view>
</views>

```

Code Fragment 5.6 – An example XML document that represents data for elements onscreen

2. *The ability to represent lists of data.*

The client requires lists of data to be able to populate drop-down menus and check-box lists. A simple model can be used to represent this data. A `lookupLists` tag can be used to group all the lists. This tag contains multiple `lookupList` tags, each one representing a single list. A `lookupList` can contain many `lookupItem` tags. The `lookupItem` tag is designed to be able to retain a displayable value and a key. This is the typical structure required by an onscreen list element as the user requires a readable value and the system requires a key to process the selection.

Code Fragment 5.7 gives an example of the XML structure described above. This XML describes two lookup lists: one contains items relating to a well and the other relating to wellbores. Each list contains a number of data items. Each data item contains a label attribute for display purposes and a value attribute for the key required by the application.

```

<lookupLists>
  <lookupList id="wells">
    <lookupItem label="Test Well 1" value="well.uid1"/>
    <lookupItem label="Test Well 2" value="well.uid2"/>
  </lookupList>

  <lookupList id="wellbore-well.uid1">
    <lookupItem label="Wellbore 1" value="wellbore.uid1"/>
    <lookupItem label="Wellbore 2" value="wellbore.uid2"/>
    <lookupItem label="Wellbore 3" value="wellbore.uid3"/>
  </lookupList>
</lookupLists>

```

Code Fragment 5.7 – An example XML document that represents lookup lists

3. *The ability to represent tables of data.*

The client requires tables of data to be represented within the XML. Typical tables consist of column headers and rows of data, each row containing relevant data to its header. This will be represented in the XML by splitting the data into a header and data section. As a table will be an element on screen it will be represented by the element tag with the type attribute set to "table". The table element will contain two child tags; a data-labels tag and a data-items tag. Data-labels represent the header information required for the columns. It will contain a text attribute for the name of column and a linked data-field so that a data item property can be matched to correct column. Data-items represent the data section of this element. This tag can contain multiple data-item tags where each tag represents a row of data for the table. A data-item can contain multiple property tags. Each property tag represents a single cell in the table. It contains a name attribute that allows the property to be matched to the data-field in the header and a value attribute that contains the value that the cell should contain.

Code Fragment 5.8 gives an example XML of the table structure. This XML describes a table of well-site data. It contains five headings and two rows. It is clear from this figure which columns the data properties match to and would be a simple matter for the RIA client to parse this data.

```

<element id="traj-details" type="table">
  <data-labels>
    <data-label text="Md" datafield="md"/>
    <data-label text="Tvd" datafield="tvd"/>
    <data-label text="Record Time" datafield="dTimStn"/>
    <data-label text="Incl" datafield="incl"/>
    <data-label text="Azi" datafield="azi"/>
  </data-labels>
  <data-items>
    <data-item name="Station 1">
      <property name="md" value="200"/>
      <property name="tvd" value="300"/>
      <property name="dTimStn" value="2007-07-01 10:30"/>
      <property name="incl" value="20"/>
      <property name="azi" value="30"/>
    </data-item>
    <data-item name="Station 2">
      <property name="md" value="400"/>
      <property name="tvd" value="500"/>
      <property name="dTimStn" value="2007-07-01 10:45"/>
      <property name="incl" value="40"/>
      <property name="azi" value="70"/>
    </data-item>
  </data-items>
</element>

```

Code Fragment 5.8 – An example XML document that represents table data

4. *The ability to represent labels of onscreen elements.*

This requirement stems from the idea of the internationalisation of the application. If the client was designed with a defined interface in a single language, with the labels hard-coded into the application, it would make client company customisation and language translations extremely difficult. The main way to solve this problem is to leave the label population until the runtime of the application. This requires the server to supply a list of labels when a client first connects so that the client can populate its screen. The XML required to encapsulate this data will be quite simple as only a small amount of data is required. The XML will consist of a main "labels" tag and multiple "label" tags. The labels tag is the grouping element that contains a list of all the individual labels onscreen. Each label tag represents a single label on screen and contains two attributes; the key and value.

The key attribute is made of the ID's of elements used in the RIA client.

key="<view-id>.<element-id>"

The section before the "." is the ID of the view that the label is located in and the section after is the ID of the element that the label should be applied to. This allows the XML processing unit in the RIA client to find the element and update its label accordingly. The value attribute is the actual value that should be displayed in the label.

Code Fragment 5.9 gives an example XML of a labels excerpt. This XML describes the labels required for the login screen.

```

<labels>
  <label key="login.message" value="Welcome, please login"/>
  <label key="login.username" value="Username"/>
  <label key="login.password" value="Password"/>
  <label key="login.loginbutton" value="Login"/>
</labels>

```

Code Fragment 5.9 – An example XML document that represents onscreen labels

5.1.1.3 Handling Extendibility

The nature of client development adds the requirement to cater for the ever changing needs of a client. XML was chosen as the data interchange language because it can be easily updated to allow future development and basically solves this need. However, this requirement is also tightly coupled to the XML processing in the application as the server must be able to cater for updates and changes without breaking the server or client XML processing. In order to achieve efficient processing coupled with the extendibility required, it would be beneficial to use technology that could be reused as the XML requirements change. Using JAXB, Java code can be automatically generated from an XML schema allowing new code to be created each time new requirements are added to the system.

System Core Extendibility

The core of the system will use an XML configuration file to define the business logic and allow updates and new features to be applied easily. It also allows custom end user configuration to be applied without any changes to the application code. The business logic will consist of three main parts; the configuration, services and query types. The configuration section will consist of the linkage elements required to setup the server and to interact with a client. The services section will include all the available WITSML servers that the system is setup to interact with. The query types section will define the different types of WITSML query that can be applied to a WITSML server. These queries will be used to check if new real-time is available and will contain parameters to be able to extract real-time data.

The server configuration will need to include the following elements.

- *Identity* – This section will define the details about the server. It will include elements like the language the system is setup in, a unique name that is

assigned to the server and the current version number. It will be used to initialise the server and to supply identity information to the Flex client.

- *UID-mapping* – This section will help process WITSML documents by creating a mapping between the WITSML object names and the field that uniquely identifies the object. It will be used by WITSML processing classes to help unique identifier recognition in WITSML queries.
- *Class-mapping* – This section defines the linkage between a WITSML object name and the Java class that represents it. It will be used by WITSML processing classes to help convert WITSML documents between XML and Java.
- *Client-mapping* – This section defines a mapping of the JBoss session beans available for a client to invoke. It maps the bean to an ID that will be hard-coded into the Flex client. This allows server processing to be changed by altering the bean that an ID is mapped to without having to alter the client. By using this approach the system will be easier to maintain as it decouples the client executions from the server classes.

As this project centres on retrieving and viewing data, the data sources must be defined in the configuration. This allows a connection to be established to other WITSML servers and data to be extracted. This section will need to define the address and authentication details of an endpoint as well as a user-friendly vendor name and the type of query required by the client. This definition is depicted in Code Fragment 5.10.

```
<services>
  <service vendor="IDS Test Site" username="idsadmin" password="idsadmin"
           queryType="DepthBased">
    http://192.168.1.69:8080/rtv_source/services/WMLS
  </service>
</services>
```

Code Fragment 5.10 – An example xml document of defined services

The server must define the type of query required for the services available. As the API of WITSML does not define a standard method of querying for updated data, it means that companies may choose to implement different solutions to this problem. The different implementation defined in this XML document will be researched from

companies who use the standard to gain an insight into how real-time WITSML updates are performed in the industry.

The structure of the query types will be based around the information required by the system to query a WITSML Server. Code Fragment 5.11 gives an example of this structure. It will include many query-type tags where each one defines the setup required for a company. It will also include mappings to JBoss session beans to allow easy reconfiguration of the query.

```
<query-types>
  <query-type name="DepthBased">
    <query-object witsmlname="trajectory">
      <queryTypeClass>TrajDepthQuery</queryTypeClass>
      <customQuery>JMSTrajectory</customQuery>
      <resultProcessor>TrajectoryResultProcessor</resultProcessor>
      <interval>60</interval>
      <conditional-field name="mdMx" type="Increase"/>
    </query-object>

    <query-object witsmlname="log">
      <queryTypeClass>LogDepthQuery</queryTypeClass>
      <customQuery>JMSLog</customQuery>
      <resultProcessor>LogResultProcessor</resultProcessor>
      <interval>4</interval>
      <conditional-field name="endIndex" type="Increase"/>
    </query-object>
  </query-type>
</query-types>
```

Code Fragment 5.11 – An example XML document of a query type

1.14.2 WITSML Client Design

5.1.1.4 WITSML Query Handling

The WITSML query language is capable of querying a large set of WITSML objects. In order to encapsulate a number of these objects, a Java interface will be created to represent the functionality required by a generic WITSML object query. This will create a base design that can be extended in multiple Java classes. These classes will represent a query interface for each type of object and fulfil Requirement 1 of the WITSML Client requirements.

WITSML Query Interface

The interface will provide methods to produce queries from the following types of parameters.

- *Single parameter* – The unique identifier of the object can be used as a parameter to be able to create a query for a specific object. This requires a mapping of the UID field name for each object to be stored in the configuration so the correct field can be set with the value. This field name can then be matched with the UID value and used to create the query.
- *Multiple parameters* – Parameters will be placed in a map data structure that will contain the name of the field as the key and the value of the field as the value. This allows fields in a query to be populated according to its name. It also introduces a limitation on the parameters as field names are not unique within WITSML queries and specifying a field name does not define which object it is attached to. In order to resolve this problem, another method will be added to support a custom query object. This object will contain the name of the fields to populate, the values to populate and the child objects required. Child objects will be represented by the same custom query object as its parent to allow a reusable structure.

WITSML Query Generation

WITSML queries follow the same structure as the documents that represent the data. This fact can be exploited when designing a query generator as the data object can be used as a template for the query. As the project is using the JAXB it will automatically generate Java classes that represent the WITSML data objects. The generated classes contain Java 5 annotations. Code Fragment 5.12 shows an example of an annotation that marks the class attribute `name` as a required XML element. This assignment originates from the schema as the XML element that is mapped to `name` is required for schema validation.

```
public class ObjWell {  
  
    @XmlElement(required = true)  
    protected String name;  
    ...  
}
```

Code Fragment 5.12 – An example excerpt from a generated JAXB class

Using a specialised library these annotations can be read from the Java classes to give additional information about the simple and complex elements held within the class. Using this information the query generator component can determine the Java object type used to represent the attribute or element and populate it with data. The data supplied to the query will either originate from a parameter supplied by the RIA or be generated by the system as dummy data to ensure every field is populated. This produces a complete and valid WITSML document. After the document is produced it will then be processed using an XSL transformation file to remove the dummy data but leave the RIA parameters as query fields (Ogbuji, 2006). This process is depicted in Figure 5.8. It shows the use of JAXB classes to produce WITSML documents and how these documents use XSL to create the WITSML Query.

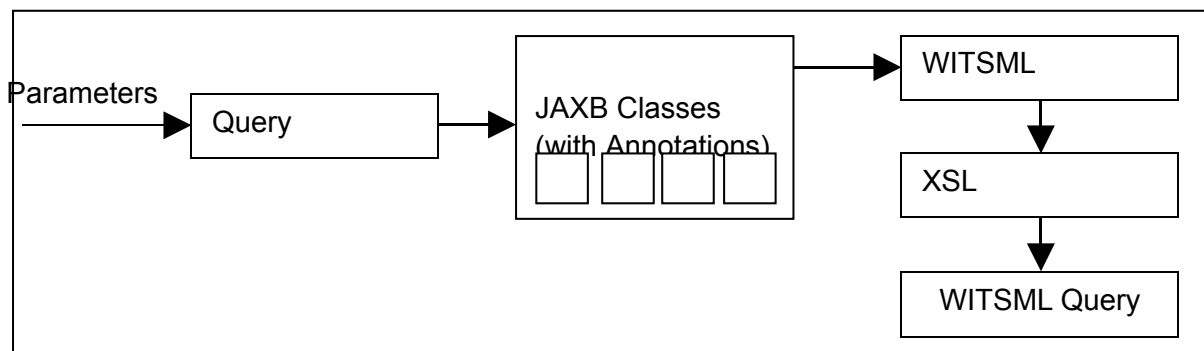


Figure 5.8 – A diagram of the operations required to produce a WITSML query.

WITSML Query Types

A number of different types of query will be needed in the system.

- *Full Query* – This query will include all elements possible in the document. It will contain fully populated child objects and fields. It will be used to extract all the data about a WITSML object.
- *Minimal Query* – This query will include only elements that are defined as required in the schema for a specific WITSML object. It will be used to find all the available objects of a specific type on a WITSML server.
- *Custom Query* – This query will include elements that are required by the schema and any extra fields that are defined programmatically. This allows queries to be easily customisable and refined to the user's requirement for the

current section.

Each of these types of query will be represented by a method in the query generator component.

5.1.1.5 The WITSML SOAP Client

As the WSDL is defined in the WITSML standard it can be used to automatically generate an implementation of a web service client. The business logic of the query will be contained within wrapper classes that define the WITSML object and fields required for the query. It will use the query generator component to create a WITSML query and invoke the relevant method on the client to retrieve the data.

SOAP Client Configuration

In section 1.14.1 the requirements of the system were analysed to create the elements that would be included in the core configuration XML; one element was the query type section. This element can also be used to connect various WITSML objects together by specifying the fields that link them. This allows the configuration of the linkage to be updated as the WITSML standard evolves over time.

SOAP Client Job Scheduling

In order to provide a usable SOAP client, the system must be able to re-run queries to update the data requested by the user. This can be achieved by using a job scheduler to schedule jobs that query a WITSML server at set intervals. Using Quartz, a generic job interface can be created that encapsulates the main operations required for a query. This interface can then be implemented for each type of WITSML object customising it dependent on the requirements of the object.

In addition to the implementation the query job will require a number of variables in order to execute the query; these variables are described in the following list.

- *Interval* – This defines how frequently the job should be executed on the system and will be specified in seconds as this is the lowest denominator required.

- *Custom Query* – This defines the name of the Java class that will be used to check if data has been updated on the server. The job will use the JBoss naming system to lookup the class and invoke a standard method defined on the interface. This method will only query selected fields to minimise the load on the server.
- *Query Type Class* – This defines a Java class that will be used to invoke a full data query designed to extract the newly published data from the WITSML server. The class will query for all the fields that the server needs to be able to successfully update the RIA client.
- *Result Processor* – This defines a Java class that is used to extract and format the values returned from a query so they can be published to a subscriber/publisher topic. As each WITSML object is structured differently, a custom implementation of an interface will be required to process the result. The job will use the JBoss naming system to lookup the class.

In addition to these variables the job must also contain a reference to the Subscriber/Publisher topic so data can be efficiently passed from the server to the RIA client. This will require the topic to be created before the job is initialised to ensure the job has an endpoint it is publishing data to.

5.1.1.6 Handling Data Persistence

This system will use the persistence module to create a link between a MySQL database and the business logic processing. As the system will be using Java objects to represent WITSML documents the same objects will be used by the persistence module as the XML processing code. This would simplify the design as there would be no need to convert between different classes to be able to save the data. In order to achieve this, both the XML annotations and persistence annotations need to be included in the Java class.

Code Fragment 5.13 shows an excerpt from an example persistence/JAXB Java class. The top two annotations are required by the XML processor to give additional information about the structure and elements in the XML document. The bottom two

annotations are required by the persistence module to define the class as an entity bean. These annotations do not interfere with the each other as the relevant modules only read their own annotations.

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "obj_trajectory", propOrder = {
    "nameWell",
    "nameWellbore",
    "name",
    "objectGrowing",
    "parentTrajectory",
    "dTimTrajStart",
    "dTimTrajEnd",
    "mdMn",
    "mdMx",
    "serviceCompany",
    "magDeclUsed",
    "gridCorUsed",
    "aziVertSect",
    "dispNsVertSectOrig",
    "dispEwVertSectOrig",
    "definitive",
    "memory",
    "finalTraj",
    "aziRef",
    "trajectoryStation",
    "commonData",
    "customData"
})
@Entity
@Table(name="trajectory")
public class ObjTrajectory {
    ...
}
```

Code Fragment 5.13 – An example excerpt from a JAXB Java object

The main benefit of this solution is the abstraction provided in the processing of WITSML data. As a WITSML document can be loaded from the database or retrieved from a WITSML query in the same representation the processing that is required only needs to handle data in one form.

Persistence Management

The persistence module provides access to a managing object that is capable of all the functions required for database interaction (create, read, update and delete). This allows the business logic to save and reload data as the system performs static and streaming operations. The managing object also supplies advanced loading facilities so that data can be loaded using a query language. This language is very similar to other SQL constructs and provides similar features to select and order data using

parameters. This feature can be used to load data for an RIA client using the parameters it has passed to the server.

1.15 Client Design

The client will be developed in Adobe Flex 2 and will use components that exist in the toolset provided where possible.

1.15.1 User Interface Design

5.1.1.7 Client Communication

The HTTPService provides the ability to invoke HTTP calls at runtime and can also add parameters onto the request so that additional information can be passed to the server. It also allows program methods to be attached to it in order to listen for events that occur. This allows custom code to be used to catch the response from an invoked call.

5.1.1.8 Handling XML Data Content

The Flex client will be required to read XML documents sent from the server and extract the data to populate onscreen elements. A class will be implemented to catch the document returned from the server. It will use the E4X extension to generate Flex objects that represent the document (Morearty, 2007). These objects will then be processed using a series of manager classes that will extract and update onscreen components. Figure 5.9 depicts the managers required to process each different topic of data. This will create a modular design and allows it to be easily extended for future development.

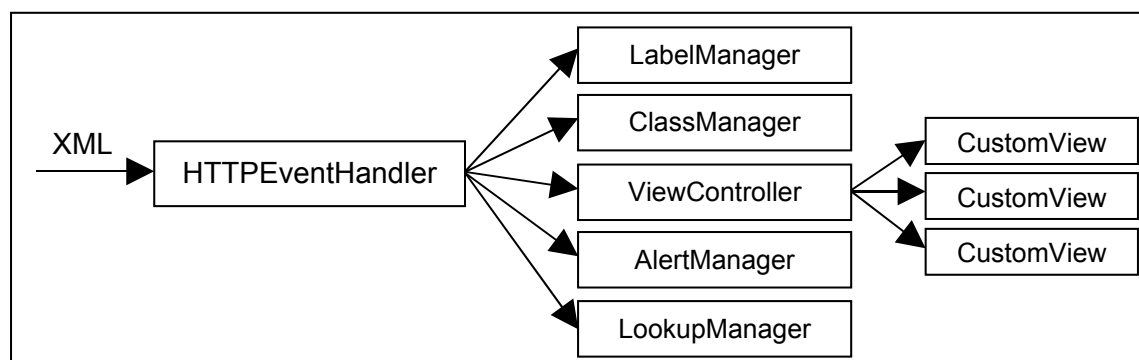


Figure 5.9 – A detailed breakdown of the XML parsing used on the Flex client.

Each manager class will extract a specific topic from the XML.

- *LabelManager* – This manager will extract any labels that have been passed from the server. It will update each component label by using the ID included to find a reference to the displayable object and once obtained, update the relevant attribute.
- *ClassManager* – This manager extracts information about the JBoss class mapping defined for the server. This mapping tells the client which class to add as a parameter on the HTTP request so the correct class is used to process the data.
- *AlertManager* – This manager handles messages sent from the server that inform the client of a specific event. This could be an automated event that the client may need to react to or a clarification/error event that the user may need to see and interact with.
- *LookupManager* – This manager extracts lookup values from the XML document to build up a collection of lists that are required in components like combo-boxes.
- *ViewController* – This manager handles the data content from the server. Code Fragment 5.6 shows the structure the data will be in when it is received by the client. This structure allows the data to be split up into different views with each representing a different screen. The processing of the data will be split up in a similar manner to increase the extendibility of the system. The manager will delegate the processing of each view to a custom class that will in-turn process the data accordingly. Each custom view class will be configured to extract and update specifically named elements.

Sectioning Data into Different Forms

The data received from the server will need to be separated and rendered into different onscreen components. The following components will be supported:

Tables

The user interface will contain a number of groups of data that need to be display in a table like structure. Table data is represented in Flex through the use of the DataGrid

component. This component consists of a header and a data section. The header section consists of column headings that label type of content held in the column. These are basic strings that can be defined at runtime. The data section is mapped to a collection object held within the Flex application. Each column is mapped to a specific attribute of each entry in the collection object; mapping an object attribute to a column. Each entry in the collection is displayed onscreen as a row in the data section.

The structure of the knowledge bubble data will be contained within a DataGrid component and placed inside a Canvas component to separate it from other elements on the screen. A Canvas is a layering component similar to a "div" tag within HTML. It allows its position and size to be changed without modifying the elements held within it. This allows the bubble to be expanded or contracted dependent on the amount of data that needs to be displayed.

The text window will contain a table of data on a selected topic related to the knowledge bubble data. A number of topics will be available for the user to select. For the purposes of this system, the fields available within a topic will be defined by the developer and cannot be customised by the user. When the knowledge bubble is updated this section also invokes an event to update its data accordingly.

2D Graphs

A number of different 2D graph types are available in the Flex Charting package. For the purposes of this project only the line chart implementation will be used. This is to limit the scope and size of the system. The graph is directly linked to its data series and so when its series is updated it automatically renders the new point. This is ideal for this system as updates in real-time will be reflected automatically on screen.

The graphs in this system will be customised using the XML content sent back from the server. It will define the axis labels and the range used as well as pre-populating any data that is available about the current topic (i.e. if a real-time stream began a few minutes ago it would load previous data before beginning to render new data).

3D Graphs

The Flex Charting package does not contain any implementation of 3D graphs. After research into this topic two main options were found for available implementations of this component; a commercial product ILOG Elixir (beta) or a custom graph implementation library by Nihit Saxena (Saxena, 2006)

ILOG Elixir

This product is a professional graphing extension for Adobe Flex. When research was first carried out the product was at the beta stage of development and allowed a free trial to be downloaded. After experimentation with the trial, it was decided that the sponsoring company did not want to pursue this option as the commercial license was too expensive.

Custom Implementation

This implementation was a standalone library created by an employee of Adobe to experiment with the capabilities of Flex. It contained implementations of three 3D graphs, a bar chart; pyramid chart and a line chart. This system requires a 3D Line Chart to represent wellbore trajectory data. From the samples provided on the developer's website the 3D graph available from this library would suit this purpose.

The library is freely available on the Flex developers section of the Adobe website and does not require a commercial license to be able to use it in an application. This is an important point for this component as although the sponsoring company want an investigation into the technology, they do not want to buy custom add-ons that may never be used in any application.

5.1.1.9 Client/Server Interaction

The Flex client will be designed to be as user-friendly as possible. In order to achieve this, elements onscreen will invoke events by clicking on them, giving the user clear visual interaction with the application. The knowledge bubble and 3D graph are a good example of components that require visual interaction. The 3D graph will be an interactive component that will allow its data points to be clickable. Each time the user clicks on a point it will invoke a query to the application server to retrieve detailed

data about the point selected. These queries will be generated by a method in the Flex client that will extract the required parameters from the 3D graph and invoke a HTTP call to the server with these parameters to retrieve the data. The server will then return this data in the defined XML format and the Flex client will process it and populate the knowledge bubble.

1.15.2 The Subscriber/Publisher Interface Design

5.1.1.10 Provide an endpoint for real-time data to be streamed.

For this project, LCDS will be deployed in a separate web-project inside the JBoss application server. This allows it to be used by the Flex client and not to interfere with the web-based endpoint mentioned in Section 1.11.1.

In order to create a subscription between the client and the server, the LCDS web-project must be configured to bridge the connections. As both the application server and LCDS support JMS-based communication this technology will be chosen. In order to use this technology, it requires the LCDS core configuration (which is based in XML) to be updated with the correct setting to monitor and update JMS topics from the server.

On a Flex client, an object called a consumer must be created to link to the configuration in LCDS. The consumer allows options to subscribe and unsubscribe to feeds of different topics allowing the user to change the topic if they wish to. Once the consumer makes a subscribe call it informs LCDS that the client wants to subscribe to a topic and that it should maintain the connection. If the consumer then unsubscribes from the feed, LCDS cleans up the resources used and waits for further requests.

LCDS is a self-contained web-application that effectively manages the requests made by Flex clients. As the majority of this section will use the existing implementation, the code should not generate memory leaks and will clean up all objects created. This will allow the system to be streamlined and work efficiently.

Implementation Issues

This section contains a discussion of the issues faced when implementing the design discussed in the previous section. It will focus on the main challenges of the implementation and describe the decisions taken to reach a solution.

1.16 Server Implementation

The implementation of this component contained a number of issues that arose from unfamiliarity with the application server. The three main issues faced were the JMS Topic configuration; the generation of WITSML queries and the persistence of WITSML data.

1.16.1 RIA Interaction Implementation

5.1.1.11 Server Communication

After the initial investigation into JBoss it proved hard to find documentation about the latest version of the application server. This was mainly due to the fact that the developers were updating the server to fulfil the Java 5 EE specification. This meant that some elements had been upgraded while others were still in development. The latest version of JBoss had altered its setup from using standard XML configuration to using Java 5 annotations. This changed a lot of the basic configuration for adding custom configured classes into the system and due to unfamiliarity with the application server it presented a large problem when implementing this component. In order to resolve this problem a portion of time was spent researching tutorials and sample code to discover the correct way to use the annotations and become familiar with the application server.

The design for this feature required a class to be implemented to control the JBossMQ component. To achieve this, the class must be able to access a reference to a controlling element in the component. In version five of JBoss, JBossMQ was planned to be replaced with a new integrated component customised for the application server. When this technology was investigated the server was still using JBossMQ and it was decided at this point that this component would be used instead of a newly integrated

component. Therefore version 4.2.1 of the application server was selected. This version contained some upgrades to conform to the Java 5 EE specification but had not changed the JBossMQ component. After the initial decision of using JBossMQ, the component was investigated to find documentation on its usage and programmatic interaction. From the documentations it was discovered that in order to create and destroy JMS topics within JBossMQ the class would require a reference to the Destination Manager class. This class manages all the connections available and would allow an entry point into the component. Through a number of tutorials a method for obtaining a Destination Manager Managed Bean was discovered. This managed bean had direct access to the internal functions of the component and could be used to control it for this system. However, the discovered method used JBoss specific functions to obtain the managed bean. This would mean that if the code was used in a different application server this component would not work correctly. Code Fragment 6.14 shows the code required to obtain this bean. The use of the `locateJBoss()` method tightly couples this component with the JBoss server. After discussions with the sponsoring company it was decided that as this project was partly an investigation into the JBoss application server using JBoss specific code was not an issue for them.

```
// find the local MBeanServer
MBeanServer server = MBeanServerLocator.locateJBoss();
// Get a type-safe dynamic proxy
mbean = (DestinationManagerMBean)
        MBeanServerInvocationHandler.newProxyInstance(server,
        objectName, DestinationManagerMBean.class, false);
```

Code Fragment 6.14 – The code required to obtain a reference to the Destination Manager

1.16.2 WITSML Client Implementation

5.1.1.12 WITSML Query Handling

The design for this component mentioned a specialist library required to read JAXB annotations from the Java classes. This library is known as the JAXB-Reflection library and was written by the developers of the JAXB technology. This initially caused a few issues as the structure of classes used in this library were unlike any other type of reflection seen in Java and was heavily customised to JAXB. This required a period of time to be spent learning the objects used in the library and how they interact with the JAXB structure.

After an investigation into JAXB reflection the following points were discovered.

- The library can represent a JAXB object as a model using the `RuntimeTypeInfoSet` object. This object contains information about all the elements and attributes inside the XML document the JAXB object represents.
- The library can extract information about a selected class using the `getTypeInfo` method on the model and represents this data using a `RuntimeClassInfo` object.
- The `RuntimeClassInfo` object contains a number of properties that represent the elements and attributes inside the XML document. These properties can be extracted and are represented by `RuntimeElementPropertyInfo` and `RuntimeAttributePropertyInfo` objects. These objects can be used to extract information about the element/attribute.

With this knowledge the implementation of this section continued according to the design. However, the design of this component did not include the handling for some of the more difficult situations that occurred when trying to generate complex queries. A few problems arose when this section was implemented and these problems are listed below:

Complex Element Problem

In order to generate a WITSML query, a blank WITSML document must be produced with default values. As WITSML documents can contain complex elements these must be handled when a query is generated. A WITSML document can either contain multiple or singular complex elements of the same type. These elements are represented in the JAXB code as a singular Java object or a list of Java objects. In order to create a robust query generation component a generic way of handling complex elements was required. This presented a problem when implementing this component as it would need to handle many different types of complex elements.

The solution to this problem involved splitting the complex elements into two main types: Singular Complex Elements and Multiple Complex Elements.

Singular complex elements are similar to the parent element that contains them. Structurally they are almost exactly the same as their parent element, so much so that parent elements can be regarded as complex elements. As a basic WITSML document (a document that only contains simple elements) could be generated using a method to iterate through its elements and attributes, the same method could be applied to singular complex elements. This allows the code to be implemented in one small method that is repeatedly called to generate each complex object. Using a recursive method this can be achieved and allows a WITSML document and all its singular complex elements to be created.

Multiple complex elements can be regarded as a list of singular complex elements. Therefore, to be able to handle these elements the query generation code will monitor the properties of each complex element to discover if it's a list. If it is a list it will generate a singular complex element using the code discussed previously, add the element to a newly created list and then set this as the value for the complex object.

Enumerations Problem

A WITSML document can contain a number of the simple elements that are restricted to an enumerated list of values. These enumerated values are listed in the schema definition. When JAXB classes are generated the enumerated values are copied into Java 5 enumeration classes and a reference to this class is then placed in the JAXB object. This limits the range of values possible for the simple element.

The query generation process needs to be able to iterate through all the fields in a WITSML document in order to generate a query. A problem occurred when processing enumerated fields as they could not be handled by the complex element code described previously or primitive type handling.

In order to solve this problem, custom handling had to be added into the field processing code to handle the enumeration class type. This required the possible enumerations to be extracted from the JAXB class and for one value to be selected from the list. This value was then assigned to the JAXB object and produced the required result.

Unit of Measures Problem

A number of elements in a WITSML document can contain values that are measured in a specific unit. This is expressed in an element by an attribute called "uom". This attribute specifies the unit the element is measured in and is required when generating this query. When initial queries were generated using this system, this attribute was not included and invalidated the query. This problem needed to be solved to be able to query for any measurable element.

After examination of the enumerations it was found that the first value in each enumeration was the SI unit. It was then decided that the system would only use these units making all values that were cached to be saved in SI units. The problem was solved by updating the field processing code to use the enumeration code with the "uom" attribute and populate it with the first value in the enumeration.

5.1.1.13 Data Persistence

The design for this section defined that WITSML data will be saved into a database through the use of the JBoss Persistence Manager. It will use Java objects to represent the rows in the database tables. As discussed in the design these Java objects will contain annotations for both the JAXB parser and the persistence manager.

Problems occurred when the annotations required different object types causing one or both of the components to fail. In most occasions the attributes used by both components use the same object type however there are a few examples where this is not the case. The two main problems found were both caused by the object representation used by JAXB and its incompatibility with the persistence mapping.

JAXB Object Problems

A WITSML document can contain a number of elements that allow extra information to be added about them through the use of attributes. When this element is transformed into Java (using JAXB) it is represented by a wrapping object that contains the value of the field and the attribute data. If the element did not contain the attributes it would be transformed into a primitive value simply containing the

value of the field. This primitive value could easily be saved by the persistence manager but as the element is represented by an object this causes an issue for the persistence manager.

The persistence manager requires an object to either represent a table or be embedded within an object that does map to table. When objects are embedded, the variables in the object can be mapped to attributes within the parent object. This allows the values to be saved without requiring additional tables. In this circumstance it would not make sense to map the wrapping object to a table as it simply contains a value and meta-information. Therefore to solve this problem the object was embedded inside the class and mapped to attributes. Code Fragment 6.15 gives an example of an embedded depth object. It shows the element `mdMn` has been represented with an object called `MeasuredDepthCoord`. This is a wrapper class that contains the value, its UOM and its Datum value. The object has been embedded into the parent object, mapping each attribute of the wrapper class to an attribute in the parent class.

```
@Embedded
@AttributeOverrides({
    @AttributeOverride(name="value",
        column=@Column(name="mdMn", nullable=true)),
    @AttributeOverride(name="uom",
        column=@Column(name="mdMnUom", nullable=true)),
    @AttributeOverride(name="datum",
        column=@Column(name="mdMnDatum", nullable=true))
})
@Column(nullable=true)
protected MeasuredDepthCoord mdMn = new MeasuredDepthCoord();
```

Code Fragment 6.15 – An excerpt from `ObjTrajectory.java` showing the mapping required for objects.

JAXB Date Parsing Problem

Date values that are used in WITSML documents include extra data to ensure the date and time is correct regardless of the time zone used on the recipients system. This involves embedding the date/time as well as the time zone offset into all date elements. When this element is converted into Java, JAXB represents it using a custom object called an `XMLGregorianCalendar`. This object encapsulates all the data required so that a date can be converted easily between Java and XML.

The `XMLGregorianCalendar` object caused a large problem with the persistence mapping because it could not be saved to the database by embedding it or by saving it as a table. It could not be embedded as the implementation of the class is closed source which means extra annotations that are required for embedding could not be added. It could not be saved as a table as the object is unserializable. This means that the object has no direct way of being converted to `String`. Date values are typically mapped to `java.util.Date` objects in the persistence mapping as this directly links to the `Date` type in SQL.

Research was carried out to find if other developers had had similar problems with this object but no positive results were found. After an investigation into possible solutions it was decided to share the date object type between the two components by mapping the object to the database but returning the correct type to the JAXB parser. The date field was altered from being an `XMLGregorianCalendar` object to a `java.util.Date` object. This allowed the persistence mapping to work correctly as it simply mapped the date field to the SQL date field. The getters and setters methods were then modified to return an `XMLGregorianCalendar` instead of the `java.util.Date` object. This provided the JAXB parser with the object it required to generate the XML. These changes allowed dates to be loaded, saved and converted into XML.

Code Fragment 6.16 shows an example of an Operations Report object with the modified date attribute. The fragment shows the getter and setter methods for the field `dTim` which the date that represents when the report was completed. Each method has been modified to convert from or to the `XMLGregorianCalendar`. This allows the value to be stored as a `java.util.Date` object but the getters and setters to use the `XMLGregorianCalendar` object.

```
/**
 * Gets the value of the dTim property.
 *
 * @return
 *     possible object is
 *     {@link XMLGregorianCalendar }
 *
 * 
```

```

    */
    public XMLGregorianCalendar getDTim() {
        if (dTim != null){
            XMLGregorianCalendar cal = XmlAdapterUtils.marshall(
                XMLGregorianCalendarAsDateTime.class, dTim);
            cal.setTimezone(0);
            return cal;
        } else {
            return null;
        }
    }

    /**
     * Sets the value of the dTim property.
     *
     * @param value
     *     allowed object is
     *     {@link XMLGregorianCalendar }
     */
    public void setDTim(XMLGregorianCalendar value) {
        if (value != null){
            dTim = XmlAdapterUtils.unmarshall(
                XMLGregorianCalendarAsDateTime.class, value);
        } else {
            this.dTim = null;
        }
    }
}

```

Code Fragment 6.16 – An excerpt from ObjOpsReport.java showing the required implementation for dates.

1.17 Client Implementation

The implementation of the Flex Client contained a number of issues that were centred on the limitation in the technology used. The three main issues faced were the representation of data in 3D; the invocation of queries through onscreen elements and the use of a real-time data endpoint.

1.17.1 User Interface Implementation

5.1.1.14 Handling XML Data Content – 3D Graph

The main problem discovered in this section centred on the integration of the 3D graph component selected in the design discussion. When this component was initially investigated its features seemed promising and able to represent the data required. However, after a detailed investigation it was found that it was incomplete and was too heavily customised to be fit for the purpose required.

The first issue with this component was obtaining a version of the code. The developer had produced a compiled version of the code with demonstration samples that was freely available from the Adobe website. However, when this file was downloaded and examined it was found that it did not contain an implementation of the 3D Line Chart. This presented a big problem as the component could not be integrated if the code was unavailable. After discovering this, the developer was contacted to find out if a distributable version was available with the 3D Line Chart included. After a number of emails the developer forwarded a distributable copy of the code with the line chart included. This allowed the component to be integrated into the Flex client.

Once the code was integrated it became apparent that a number of features were missing that would allow the graph to be easily customised. Along with these missing features there were a number of problems that occurred due to the lack of testing in the development of the code. These missing features and problems are described later in this section. After further discussion with the developer it was found that the code used to generate the graphs was closed source with the rights being held by Adobe. This meant that the code could not be viewed, easily extended or fixed for this

project. In light of these facts further integration was abandoned but the component was left in the Flex client as some of the requirements had been fulfilled.

Limitation on Functionality

When the graph implementation was initially investigated the sample graphs showed two category axes and one numeric axis. This sample is depicted in . The figure shows test Olympic data results for a number of countries. It lists three data series each representing the number of awards received at a set level (Gold, Silver or Bronze). As the 2D implementation of the line chart allows each axis to be defined and data series to be plotted against these axes, it was believed that the same principle could be used within this graph.

The 2D graph implementation provides customisable axes that can either be text values or a range of numeric values. Data series values can then be plotted on the graph by matching the values on the axes. When this theory is applied to a 3D graph the expected functionality would allow three axes to be configured and points to be plotted according to the axes. However, when this functionality was tested it was discovered that the third axis had been bound to the name of the data series. This meant that it could not be used as a measurable axis and so could not plot three dimensional points.

A small investigation was then started to find why the name of the data series had been bound to the third axis. However, as the implementation was closed source the results obtained were not detailed enough to find the reasoning or be able to add this functionality to the graph. From the results of the investigation it was discovered that the objects used to construct the graph could support 3D Cartesian coordinates but the way the developer had customised it had removed support for it.

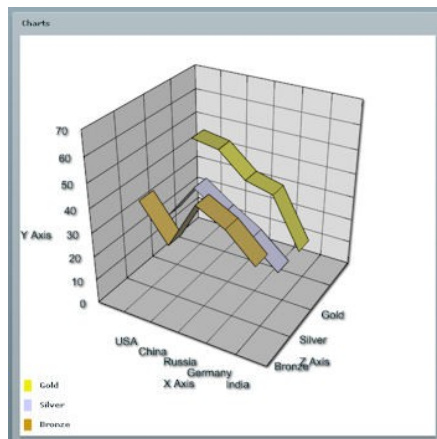


Figure 6.10 – A sample graph supplied in the 3D Flex Chart distribution

Axis Definition Problem

One of the main problems with the graph was an issue caused by the definition of the axes. An axis can contain a minimum, maximum and interval value that allows data to be spread evenly over the different planes. An initial investigation was performed on the graph before the implementation phase to investigate the rendering of a data series over different max, min and interval values. The results from this investigation showed the graph implementation acted the same as the 2D implementation. However, problems began occurring when real data was applied to the graph. Each time the graph component was loaded an exception in the program code occurred crashing the entire Flex application. This was obviously a very big issue and needed to be solved to obtain a working system.

In order to solve this problem, the data series was extracted into a separate Flex application to test the parameters of the 3D graph. After a detailed investigation it was discovered that defining an axis size and interval was causing the issue. A calculation used in the rendering graph was causing an exception to be thrown. As this code was closed source the problem could not be investigated further. As a solution to the problem, the axis definition was removed from the Flex application. This allowed the graph rendering code to assign its own values for these axes and stopped the exception occurring. However, because the axes were now defined by the system it did not represent the spread of values very well. This caused a messy list of numbers that were barely readable to be produced for the axes labels. This was not

the ideal solution but because of the closed source nature of the graph code a better solution was unavailable.

Axis Labelling Issue

One of the features missing in the graph implementation was a standard way to set the Axis labels. The 2D implementations of the graph allowed labels to be defined using attributes on the graph object. Although the 3D implementation provided the same attributes it did not update the graph with the new values. This presented a difficult problem as the internal implementation of the graph was unavailable. The solution to the problem was discovered by experimenting with various internal objects held within the graph implementation. A number of experiments were performed to test the effect of changing label related attributes and eventually the correct attribute was discovered. This allowed the graph to be labelled normally.

Sorting Data on a Different Axis

The 2D line graph implementation allowed a data series to be sorted on either axis allowing graphs to be rendered vertically as well as horizontally. This functionality was achieved through the use of an attribute on the data series. It required the attribute `sortOnXField` to be set to false therefore sorting the data on the Y field and rendering it vertically.

In order to achieve similar functionality in a 3D environment a parameter is required to inform the graph processing code which axis to sort the data series on. After searching through the available parameters in the 3D implementation the `sortOnXField` attribute was found. However, after experimentation with the attribute it was found that this feature had not been implemented and simply sorted the values on the X axis regardless of the value. As the code for sorting by a selected axis could not be altered this problem could not be solved and was simply left sorting on the X axis.

and show the 3D and 2D graph implementations rendering the same test data. Although they are both set to sort the values on the Y axis it is clear from that the 3D implementation is not sorting it on this axis.

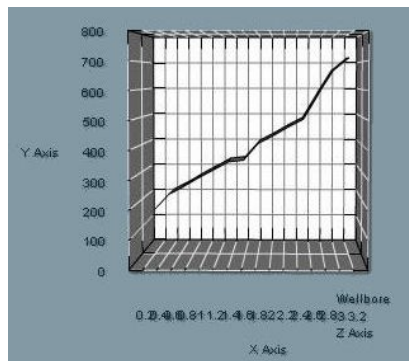


Figure 6.11 – The 3D Graph rendering test data

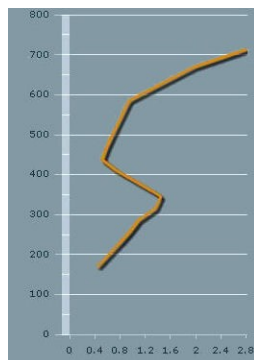


Figure 6.12 – The 2D Graph rendering the same test data as

Clearing Data Problem

The 2D implementation of the graph provided features to continue or restart the rendering of a graph if the data source changed. This allowed previous data to be saved or cleared dependent on the user's requirements. This feature was activated through a method called `hideData` that would inform the graph that it should remove the currently rendered lines and restart the rendering process. This was an issue with the 3D implementation as although it also had a `hideData` method it was not implemented. This meant that when the data source was changed, the previously rendered data was still visible on the graph. As the graph was closed source this method could not be investigated further and no solution was found.

Averaging values between points

The 3D graph implementation was designed to render three dimensional polygons between the points specified in the data series. The examples it was designed for used a categories axis for its X axis instead of a numerical one. This type of axis uses

String values instead of numbers to group results together. Using the category axis the processing code averaged the values between the categories and generated the polygons between points. However, when a numerical axis was used the same averaging algorithm caused issues for the graph rendering. The algorithm recalculated the points required to render the graph and moved them according to the easiest path required by the polygons. This caused major inaccuracy in the graph as it no longer reflected the points in the data series making the overall graph useless. There was no solution to this problem as the averaging algorithm was held within the graph implementation and this could not be changed.

5.1.1.15 Client/Server Interaction

This section is directly linked to the previous section 1.17.1. The initial design of the 3D graph component required a clickable interface that would invoke queries to update the knowledge bubble data. As the 3D graph implementation did not render the data correctly, it made it almost impossible to implement the functionality for this requirement. This was mainly due to the fact that the data provided by the 3D graph was unreliable and therefore could not be trusted to invoke queries.

This requirement was not implemented because of this problem.

1.17.2 The Subscriber/Publisher Interface Implementation

5.1.1.16 Provide an endpoint for real-time data to be streamed

The design for this section stated that the Flex Data Services (LCDS) would be used to maintain connections between the JBoss server and the Flex client. This setup was initially tested in the design of the application to find out if the data services could support the required functionality. These tests were achieved by defining the configuration settings required to communicate with the JBoss server in the setup XML files of the LCDS server. This provided successful tests that allowed data to be passed from the JBoss server to the Flex client by simply subscribing to a feed name.

A problem occurred when a connection to a JMS topic was required by a client but the name of the feed was not available until the runtime of the application. This required JMS topic configurations to be setup at runtime and maintained by the LCDS server.

After further investigation it was found that this feature had been partly implemented and that it was possible to dynamically create JMS topic configurations. The solution to this problem required a Java object to be used on the server to extend the functionality available. This object would contain references to core objects in the LCDS server and would add the topic configuration when a new topic was required.

Using this configuration a major limitation was discovered when trying to link a JMS topic to multiple Flex clients. When the configuration was added it created a one-to-one mapping between the server and the client meaning each time a different client wanted to connect to the same feed a new configuration was required. Each configuration required a large amount of code to maintain a heavy connection between the client and server. This would increase the memory and CPU usage exponentially as more clients were added. It was decided that this solution could not be used as it did not scale well and would eventually consume all the resources of the server.

As the solution produced for this problem was not viable, it was decided to investigate other possibilities. After a week of research a basic description of an alternative method was discovered on a Flex Developers Forum [LINK]. The alternative method consisted of creating two levels of communication, one between the JBoss server and the LCDS server and the other between the LCDS server and the Flex client. The LCDS server would create a singular JMS consumer per topic that would be used to retrieve data for all Flex clients. It would then use Actionscript subscriber/publisher technology to push data from the JMS consumer to the Flex client.

Figure 6.13 depicts how the new implementation will function. Data will be collected in a custom written class that uses JMS Topic Consumers to receive updates from the server. This class will contain references to the core components in the LCDS server and be able to add to the messaging streams available. When a client requires a topic connection it will communicate with this object to ensure a consumer is connected to the JMS topic required. If a consumer does not exist for this topic, it is created and initialised to receive values from the JBoss server. It will then create an Actionscript topic that will represent the clients individual feed and create a direct link between these two topics by mapping the JMS topic consumer to an Actionscript topic

producer. This will forward results to the Actionscript topic which is then picked up by the Flex client.

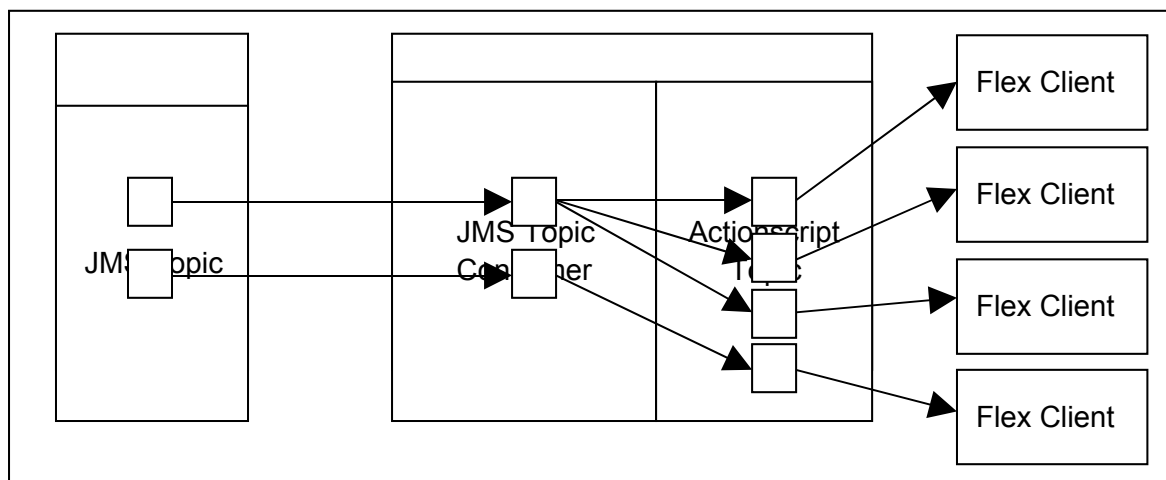


Figure 6.13 – The alternative method to topic data transfers.

Limitation in Flex Applications

A limitation was discovered in a Flex application when multiple graphs were implemented. In order to keep a graph up-to-date each graph requires a “Consumer” object to continually update it. This limitation stopped a Flex application from using more than one consumer by replacing the currently instantiated copy with the newly created one. This caused a major problem in the system as it was a requirement to dynamically update more than one graph onscreen.

The solution to this problem was found by using a different implementation of a consumer object. The MultiTopicConsumer allows a single subscription to an Actionscript topic but allows multiple subscriptions to sub-topics. A sub-topic is a stream of more specific data. It is considered as a child subject of the topic it belongs to. For this system, a sub-topic will represent a stream of data on a specific topic for a specific client. In order to use these sub-topics the topic processing code discussed earlier needs to be altered to filter results into sub-topics instead of a simple Actionscript topic. This will create a link between the JMS topic and the Actionscript sub-topic.

Figure 6.14 depicts the changes that need to be implemented to add the sub-topic functionality. The linkage requires the Actionscript topic element to be altered so that

the Flex client consumer connects to a sub-topic instead of a topic. As sub-topics replace the linkage between the Flex client and the LCDS server there is no need to create more than one Actionscript topic. This singular topic could be defined in the configuration of the server so that it immediately starts up when the server initialises. Each time a Flex client requires a new stream of data a new sub-topic will be created and linked to the existing Actionscript topic. Each time new data is delivered to the JMS topic all sub-topics linked will have data forwarded to it. The MultiTopicConsumer on the Flex client will receive data as it is published to each sub-topic it is subscribed to.

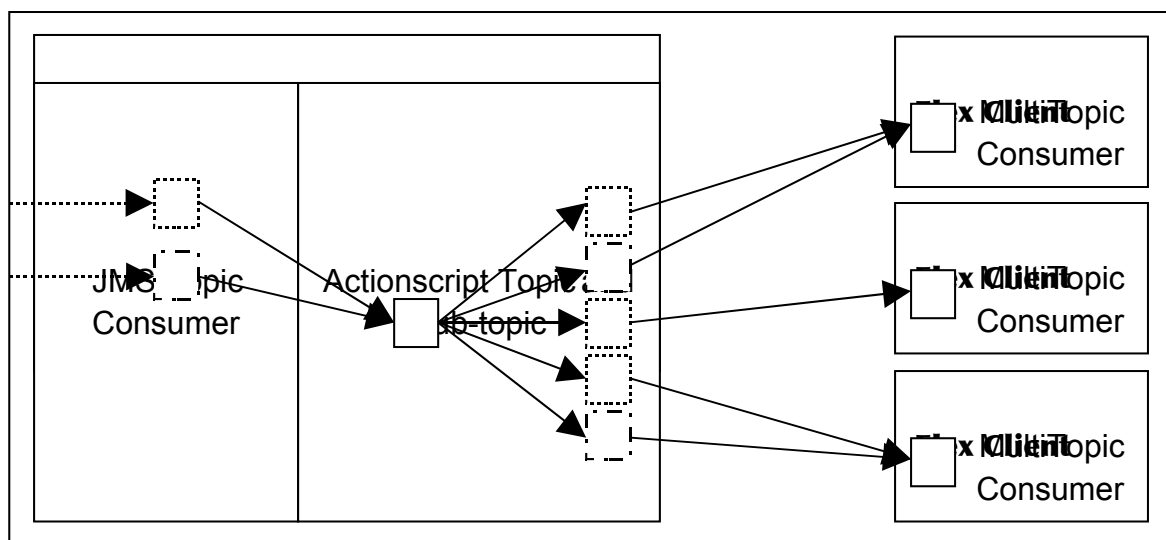


Figure 6.14 – The multi-topic solution required

This implementation allowed a single consumer to be used in a Flex application but receive multiple streams.

Evaluation

This section contains an analysis of the work completed for this project. The project will be evaluated in three parts: the client, the server and the system as a whole. The

client evaluation will focus on the usability of system. It will break the implementation of the interface into key elements and focus on the features discussed in the design section. The server evaluation assesses the abilities of the server through a discussion about performance profiling. The final section discusses the system as a whole. It will evaluate the technologies investigated and state the overall opinion of the sponsoring company.

1.18 Client Evaluation

The client consists of a number of different screens that allow the user to interact with the system. These screens include a login screen, a post-login screen, the main view and the graph view. Each screen consists of a number of elements and will be evaluated on the usability of the features it was designed to fulfil.

1.18.1 Login

The security of an application is one key element that all commercial companies require. Although security management was planned in the implementation of the system, it was not stated as a requirement. This was because the system was designed as a prototype to investigate specific technology and did not require the full functionality of a large web-based application. Regardless of this point, basic security was implemented by using the integrated security in the Java web-container. This allowed details to be passed to the web application and validated using details stored in text files. This was not an ideal solution as usernames could not be easily added or removed and passwords could not be updated without manually changing the text file. If further time had been available the login details could have been stored in a database to provide easier updating.

Figure 7.15 shows the login screen that was implemented for the system. The view contains standard features to allow a username and password to be entered and a login button to check these values with the configured users of the system. This screen is the first screen displayed when the user navigates to the URL or replaces their current display if they have left the system idle for a set period of time. The screen is very similar to other login prompts and its simple design allows a clear understanding of its use.

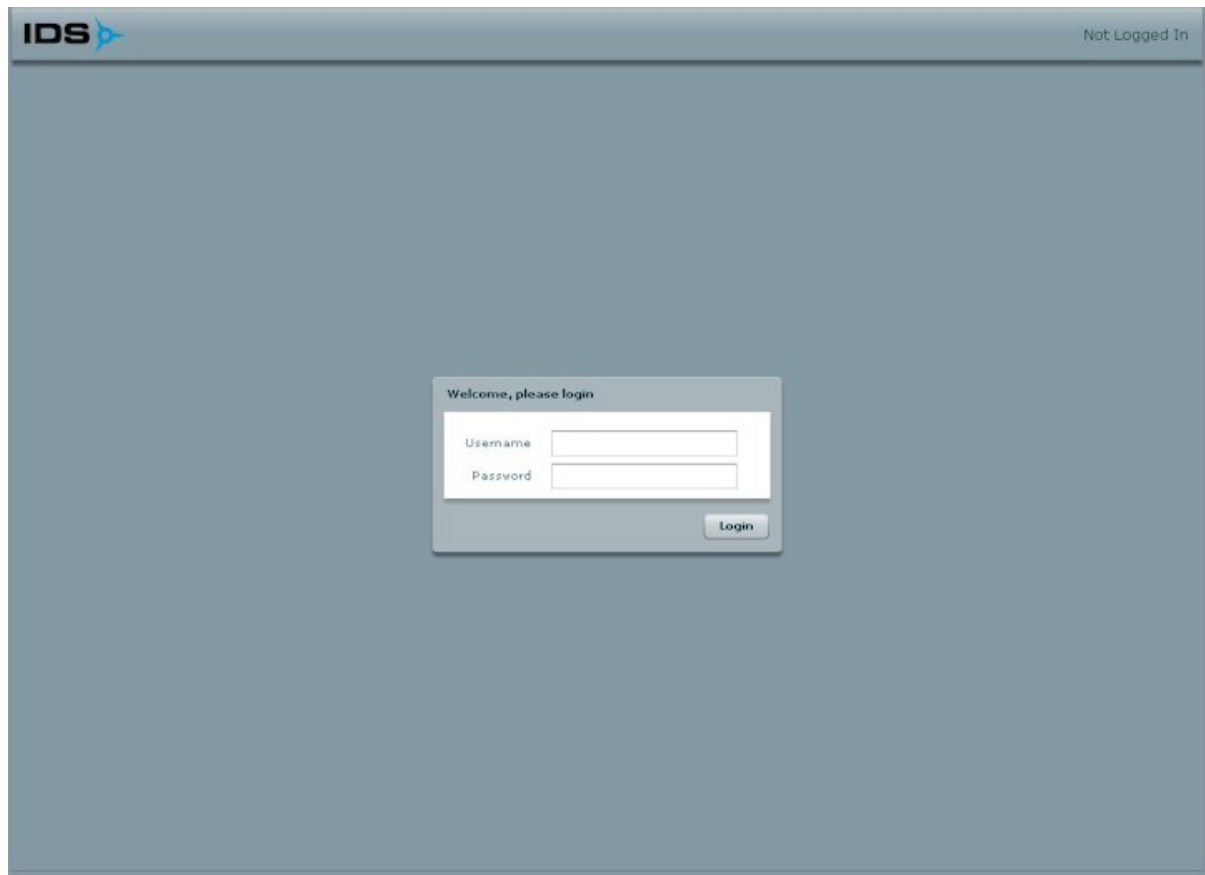


Figure 7.15 – The login screen displayed when the application is first initialised.

1.18.2 Post-login

Once the user has logged into the system they are given options to select the data they wish to view. These options consist of three cascading drop-down menus that allow them to refine which well/wellbore/trajectory they wish to view.

Figure 7.16 depicts the “Well” drop-down list that begins the selection process. Once the Well, Wellbore and Trajectory lists have been selected the system can then display information about this trajectory. By structuring the selection in this manner it provides direct linkage to the WITSML query generation on the server as these parameters are required to retrieve information about the trajectory. This gives the user a large amount of control over the application as they can select exactly what they want to view.

The user interface for this selection is simplistic and allows the user to see exactly what they need to do to view the data required. The terms used ("Well", "Wellbore" and "Trajectory") will be familiar to the clients who would use the system making each of the drop downs meaningful to them. This is an important point for the usability of the system as the users must be able to understand the meaning of elements on screen.

A bad point about this display is the fact that users will be required to select all the drop-down lists in order to view new content. In some circumstances there may only be one option available in the drop-down list that would mean the user would have to waste clicks when only one option is available. This led to a new feature being added into the system that would automatically select the first item in the second and third drop-down list when the first drop-down list was selected. This meant that the user was only required to click once to view new data. This increased the usability of the system by decreasing the amount of work required by the user.

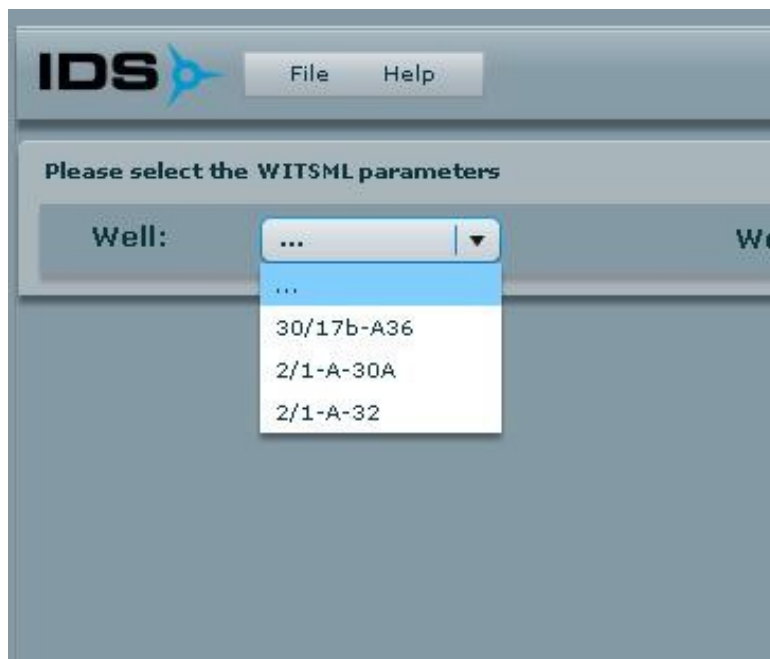


Figure 7.16 – The selection menu provided in the application to allow the user to select relevant data

One of the factors the sponsoring company wanted to investigate was the menu system available in Adobe Flex. Figure 7.17 shows the menu system implemented for this system. This menu implementation allows further menu items to be added to

customise the application for a client. Each menu item can have child menu items creating an indented tree structure. They can also have actions attached the item so that events occur in the system when the menu is clicked.

The menu system looks similar to non web-based application with titles that should be familiar to the user. This will hopefully produce a feeling of familiarity when a user interacts with the system allowing them to quickly learn the features of it.



Figure 7.17 – The main menu provided by the application

1.18.3 Main Display

The main view of the system provides the major features required by the sponsoring company. It is designed to contain most of the data encapsulated in the system and display it in the manner specified by the sponsoring company. The view is divided into three sections: the 3D graph, the Knowledge Bubble and the Text Window. Each section contains different data related to the selection criteria depicted in Figure 7.16.

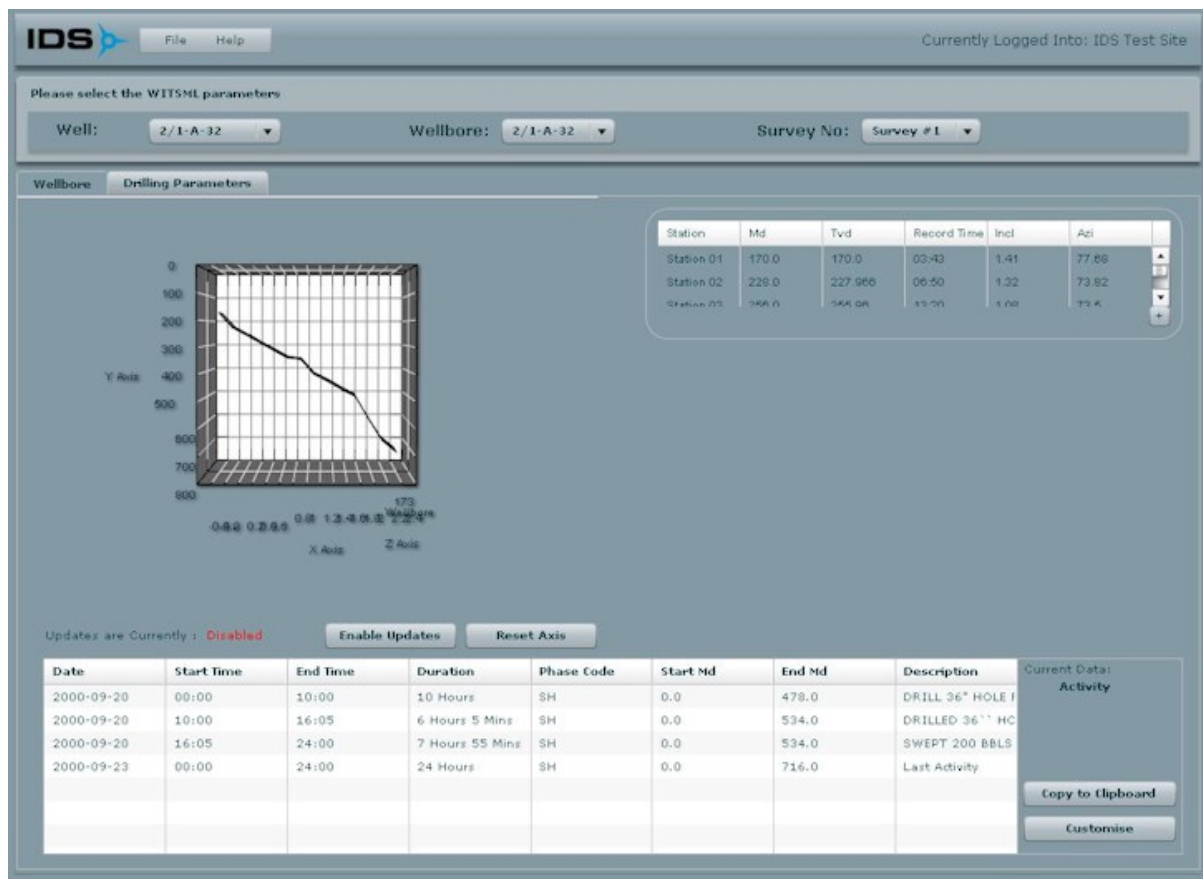


Figure 7.18 – The main view of the application

5.1.1.173D Graph

The 3D graph component is located at the top left of the main view. It contains data about the trajectory of a wellbore and was designed to represent the path taken by a wellbore when it is drilled. Due to the issues explained in section 1.17.1 this graph does not represent this, instead it measures the variation in the north/south displacement against the depth drilled. Obviously this is not ideal as the design stated a specific representation but unfortunately, due to the limitations of the current technology, this was not possible.

The interface of the graph allows the user to click and drag to move the axes and view it from a different angle. This allows the user to customise their view dependent on the data they require. Figure 7.19 shows the default angle applied to graph when the data is first loaded. The user can then click and drag any axis to change the angle applied to it. This functionality provides a useful feature that can help improve the perception of the data.

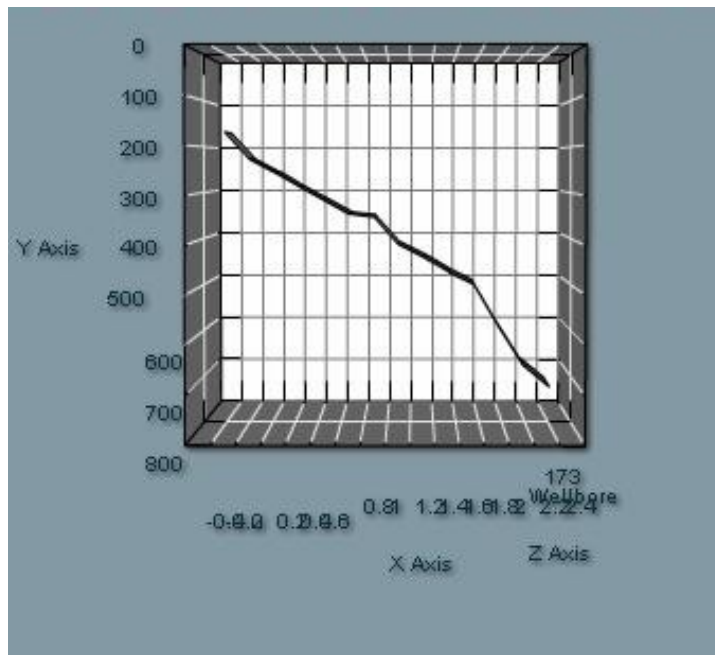


Figure 7.19 – The 3D Graph implementation from the side angle.

The 3D graph can be rotated on all three axes. Figure 7.20 shows the same graph as Figure 7.19 but the Y axis has been rotated through a 45° angle. These two figures demonstrate the functionality available from 3D graph.

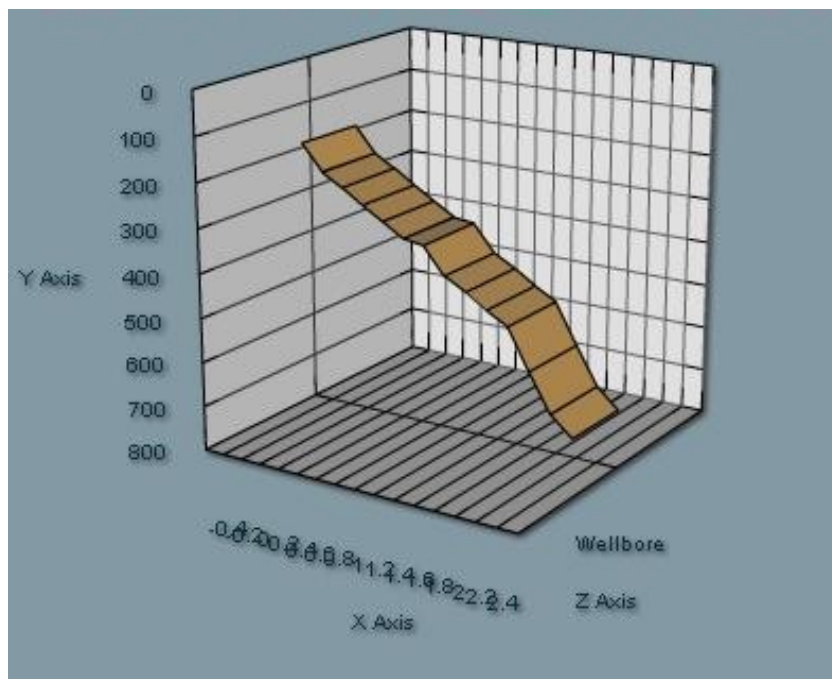
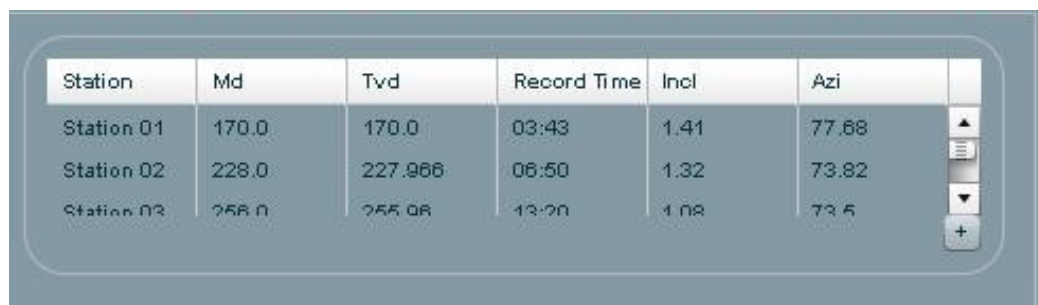


Figure 7.20 – The 3D Graph implementation viewed from a 45° angle

As this implementation is not a valid 3D representation it does not give the information that the client will be looking for. It provides some of the required features (like a rotatable axis) but due to the limitation of the implementations available the requirement could not be successfully fulfil. Since the design and implementation of this project, additional 3D graph libraries have been released for Adobe Flex. Therefore, if this project was implemented at a later date the components available may have been able to fulfil the requirement.

5.1.1.18 Knowledge Bubble

The Knowledge Bubble was originally designed as an interactive pop-up that would display additional data about points on the 3D graph. However, due to the limitations of the 3D graph this interaction was not possible forcing this component to become a basic version of what was discussed in the design. Figure 7.21 shows an example listing of data that is available in the component. It displays five attributes of each point on the 3D graph.



Station	Md	Tvd	Record Time	Incl	Azi
Station 01	170.0	170.0	03:43	1.41	77.68
Station 02	228.0	227.966	06:50	1.32	73.82
Station 03	258.0	255.08	13:20	1.08	73.5

Figure 7.21 – The implementation of the Knowledge Bubble in condensed mode.

The component can be viewed in two different modes: condensed mode and expanded mode. These modes control how much data is available on screen.

Figure 7.21 shows the component in condensed mode. This mode displays only a few lines of data and was designed so that only a small amount of information would be displayed onscreen. However, the filter required for this was not implemented because the values required could not be extracted from the 3D graph. This component therefore displays all the values in a condensed size.

Station	Md	Tvd	Record Time	Incl	Azi
Station 01	170.0	170.0	03:43	1.41	77.68
Station 02	228.0	227.966	06:50	1.32	73.82
Station 03	256.0	255.96	13:20	1.08	73.5
Station 04	287.0	286.95	17:20	0.78	73.3
Station 05	316.0	315.95	23:00	0.85	16.71
Station 06	345.0	344.95	07:00	0.47	191.75
Station 07	351.0	350.95	12:20	0.54	194.9
Station 08	411.0	410.94	19:32	0.74	173.27
Station 09	438.0	437.94	09:00	0.49	243.53
Station 10	469.0	468.94	12:40	0.49	347.29
Station 11	496.0	495.94	14:30	0.21	129.0
Station 12	586.0	585.94	23:30	0.51	347.99
Station 13	670.0	669.93	07:00	0.92	10.9
Station 14	714.0	713.93	09:00	1.26	11.46

Figure 7.22 – The implementation of the Knowledge Bubble in expanded mode.

Figure 7.22 shows the Knowledge Bubble in expanded mode and displays an expanded window the same height as the 3D graph. The window contains a list of attributes of various points from the 3D graph.

Overall this component lacks the features required in the design. It does not change its content based on user interaction with the 3D graph and this was a key feature that would have been useful to users. The lack of user interaction is mainly linked to the implementation of the 3D graph and as this could not be fixed it, had a negative influence on other components. This component could have been improved if the 3D graph features were implemented successfully. This would have caused it to be styled more like a pop-up window that would have allowed a more useful and usable component.

5.1.1.19 Text Window

The text window was designed as a container for additional data displayed in a table like form. Figure 7.23 shows the implementation of this component in the system. It displays data related to activities performed on the oil rig at the date and time the points on the 3D graph was implemented. The original design stated that different

data could be contained within this component. The window was implemented with this feature in mind by creating the table header and data based on the content received from the server. A "Customise" button was also placed on the GUI to allow the user to change the data they are currently viewing. Additional to this a "current data" label was also added to show the user which WITSML object they are currently viewing.

While implementing this feature it was discovered that a large portion of work would be required to customise different WITSML object output from the application server. It would also require an addition screen to allow the user to filter which WITSML object they want to view data on. After reviewing this discovery, it was decided that these additions would require too much time to implement and this feature was left out.

Overall the component successfully displays the data required and allows the user to see related data about the activities on the rig. Although it did not include all the functionality planned, as a prototype it gives great potential for future development.

Date	Start Time	End Time	Duration	Phase Code	Start Md	End Md	Description	Current Data: Activity
2000-09-20	00:00	10:00	10 Hours	SH	0.0	478.0	DRILL 36" HOLE I	
2000-09-20	10:00	16:05	6 Hours 5 Mins	SH	0.0	534.0	DRILLED 36" HC	
2000-09-20	16:05	24:00	7 Hours 55 Mins	SH	0.0	534.0	SWEPT 200 BBLs	
2000-09-23	00:00	24:00	24 Hours	SH	0.0	716.0	Last Activity	

Copy to Clipboard
Customise

Figure 7.23 – The Text Window implementation

1.18.4 Graph Display

The graph view provides the 2D graph features required by the sponsoring company. It is designed to contain a number of graphs that could receive streams of real-time data and render it vertically on the screen. Figure 7.24 shows the implementation of a view containing four identical graphs that each allows a stream of data to be connected to it and render the results. The user interface is quite plain and repetitive to simplify the design; this is because the system is only meant to prototype the streaming requirement and does not require a complex UI. Although basic, the components available onscreen are clearly grouped together and allow the user to interact with streams of data.

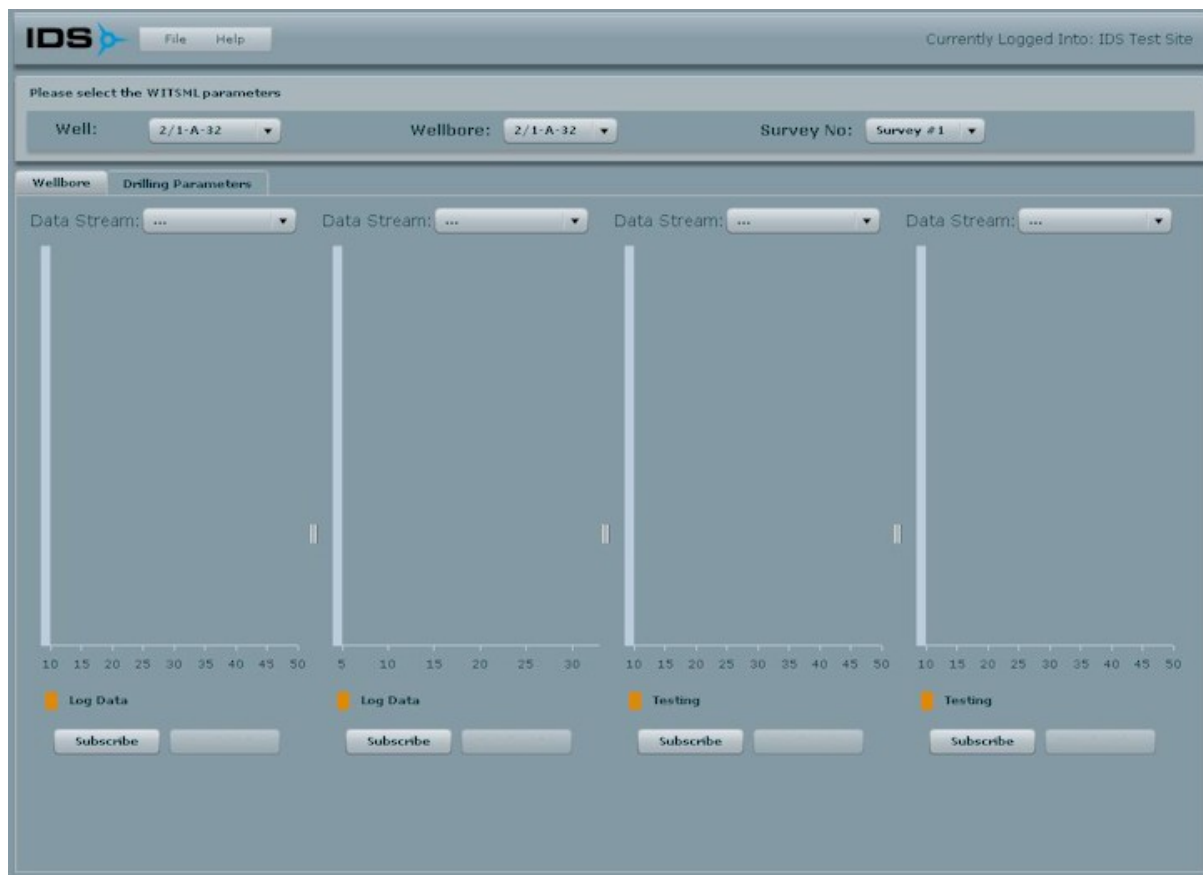


Figure 7.24 – The graph view implementation used to render streamed data

5.1.1.20 Selection of Streams

The controls available for each graph allow the user to subscribe to data feeds. Figure 7.25 shows a detailed view of a single graph on the graph view. It shows a number of

data streams that are available for the client. The user must select one of these and press the “Subscribe” button in order to prepare the application to receive a data stream. This approach allows a clear configuration for a user to allow them to begin viewing data.



Figure 7.25 – A 2D graph showing the data streams available

As multiple graphs are available in this view an individual data stream should be exclusive to each graph. However, an issue was found in the selection of data streams. Flex clients should only be able to subscribe to the same stream once however as each selection list is separate it does not prevent this from occurring. This bug was not discovered until after the cut-off point in implementation and has remained in the system for this reason. In a production version of the system the list would update dependent on the selection of other graphs.

5.1.1.21 Graph Rendering

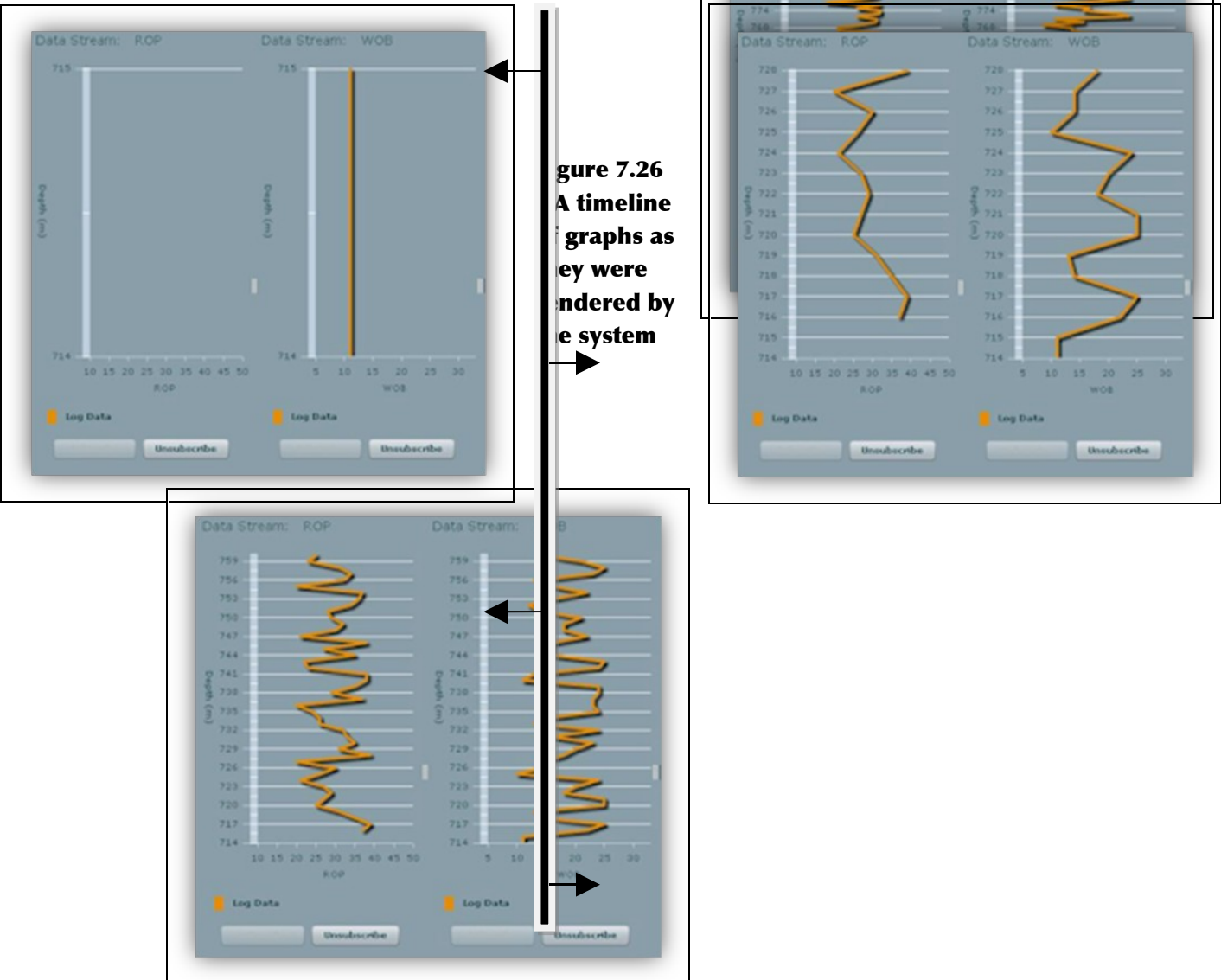
Graph rendering in Flex automatically updates when new data is added to its data source. As the subscriber/publisher automatically provides the data these components update without any interaction from the user. Once the user has subscribed to a specific topic the system automatically connects all the components required for the

stream and waits for new data to arrive. As each value arrives, the Y-Axis is recalculated to match the new highest values with a new interval. shows a timeline of graphs that were rendered in the system. Each graph shows a different maximum value measured in a different interval that was dynamically changed as new values arrived.

The speed of this component is dependent on the connection available between the WITSML server, the application server, the Flex Data Services and the Flex client. This creates a number of points of delay in the system. The processing between the WITSML server and the application server will be the slowest link as these servers will usually be based on an oil rig and require translation into WITSML. Other delays may also be introduced due to the processing and translation of the data. Although delays may occur, it will simply cause streamed values to group together and be delivered to the Flex client at the same time. This is a good redundancy feature as it ensures that the data collected from the WITSML server is eventually delivered to the Flex client. Although grouped data updates may confuse the user, the speed of the connection cannot be controlled as the location of the data source cannot be altered.

This component is straight forward to use and provides the user with powerful monitoring abilities. It successfully fulfils the sponsoring company's requirements showing the technology selected is capable of real-time data streaming.

Timeline



1.19 Server Evaluation

The server consists of a number of components that work together to perform the functionality required. Each of the server components described in was successfully implemented and although problems occurred, the overall system functioned as required.

1.19.1 Performance Profiling

An attempt was made to monitor the behaviour of the server to discover the limitations of the technology. This attempt recorded the memory and CPU usage as the server performed various functionality. In order to create an accurate simulation of strain on the system a number of Flex clients were required; each client would need to perform standard data calls and subscribe to various data streams. As a standard desktop PC is limited by the physical hardware available it would not be possible to use a high number of Flex clients on a single machine and meant that an alternative method was required. An investigation was performed on the LCDS server to determine the objects that were used by the system to link to the Flex client. It was discovered that a Flex client is directly represented by Java objects on the server and each new client creates a new set of these objects when it uses web-based functionality. This allowed an emulated client to be created using a standard Java class. This client provided all the functionality required to invoke HTTP requests and subscribe to data streams.

A number of experiments were performed using emulated clients. These experiments were based around the subscription of data streams as this was a major area that the sponsoring company wanted to investigate. Each experiment was monitored using the Task Manager in Windows to view the memory usage for the application server. The following experiments were performed:

- *Single Data Topic, Varied Number of Total Subscriptions*– This experiment involved creating a single topic on the server and gradually increasing the number of emulated clients connected to it while data is streamed. This experiment was designed as an initial investigation into memory management

for a single topic. It was seen that the memory consumption of the application server gradually increased as the number of clients increased. The server managed to balance the memory consumption of about a thousand clients before each data item received started to overload this balance.

- *Ten Data Topics, Single Subscription*– This experiment involved publishing ten topics of data on the server and allowing a set of clients to subscribe to a singular topic. This experiment was designed to test how the server managed the load created by multiple topics. The Task Manager showed that the addition of multiple topics did not place noticeable extra strain on the server. Although the memory consumption did increase like the first experiment it was not noticeably different. When the number of clients were increased to over a thousand the system became unbalanced and began to crash.
- *Ten Data Topics, Multiple Subscriptions*– This experiment involved publishing ten topics on the server and allowing a set of clients to subscribe to a random number of different topics. This experiment was designed to investigate the effect of clients subscribing to multiple topics. Although multiple data items were delivered to the clients the increase of memory used was minimal when compared to the results from the second experiment. This showed that delivering multiple data items to a single client does not affect the efficiency of processing the data.
- *One Hundred Data Topics, Multiple Subscriptions*– This experiment involved publishing one hundred topics on the server and allowed a set of clients to subscribe to ten randomly selected topics. This experiment was designed to put heavy strain on the application server. When monitoring the memory consumption it was discovered this load consumed a high level of memory but was successfully maintained by the application server in a balanced manner. This showed that the application server was scalable and could maintain a large number of topics.

Although multiple experiments were performed, the profiling tool (JBoss, 2007) used could not successfully monitor the behaviour of the system or generate sufficient

results for comparisons. This was because it required high amounts of memory to be able to capture results and as the application server was already consuming a large amount of memory there was insufficient space in the Java Virtual Machine which caused it to crash.

If further time had been available a detailed analysis could have been performed on the system by creating a more sophisticated test bed and distributing the application server. This would have provided a larger amount of resources allowing the profiler to perform better.

1.20 System Evaluation

The design shown in accurately describes the structure of the system produced. The system will be evaluated in two ways: a technology evaluation and the opinion of the sponsoring company. The technology evaluation will focus on the main pieces of technology investigated and describe the good and bad points discovered when they were used. The opinion of the sponsoring company explains what the company thought of the completed system and how it would be used in the future.

1.20.1 Technology Evaluation

5.1.1.22 JBoss Application Server

The main functionality of the system was provided by the components designed in the server section. The features available in the JBoss application server allowed many of these components to be easily created and configured without the need to write or integrate custom libraries. This was a big advantage for the implementation of the system as it meant more time could be used implementing the business logic instead of focusing on required but unrelated components.

JBoss contained a number of useful features that are built into its core libraries. One of the main advantages discovered when using the application server was the load-balancing feature. This allows the server to monitor the resources each class is using and alter its settings to balance the overall server.

The structure required for projects and classes used within a JBoss application was the main drawback when using the application server. Before the start of the project only limited knowledge was understood about Java application servers. This created a steep learning curve and meant many mistakes were made when trying to setup and develop with the application server. This problem could not be avoided as the project required an investigation into JBoss.

Overall, the JBoss application server provided key functionality to the server created. It allowed all the requirements to be successfully implemented and allowed an extendable system to be developed.

5.1.1.23 Adobe Flex

At the conception of this project the sponsoring company was investigating Adobe Flex as a way forward in web application development. It was discovered through the development of this system that Adobe Flex is a powerful tool that allows RIA's to be easily created. The language allows an Object-oriented approach to be taken when developing a front-end that in turn allows a large amount of processing code to be moved into the browser. The language allows efficient coding to be achieved with automatic cleaning of the memory used ensuring that applications do not overload the browser.

Adobe Flex is limited by the fact it requires the Flash plug-in in order to execute applications in a browser. For most users this is not a major issue however when focusing on users in the oil industry the problem of security may cause problems. As large oil companies wish to maintain a high level of security they block users from installing applications that could potentially harm the system. Some companies also block the installation of plug-ins. This is an important point to remember when developing an application for clients in the oil industry.

The technology is ideally suited for developing feature rich applications. It provides a modern look with a multitude of components to help the developer produce a piece of software the user will enjoy using.

5.1.1.24 The Subscriber/Publisher interface

The real-time elements of the server were controlled by this component. The implementation of this component proved challenging but once the infrastructure was created the system performed exactly as it was designed. The interface was ideal for the streaming of this style of data as only simple numerical values were required to be forwarded inside the system. It allowed data to be passed from an application server to a Flex client without the need for continuous queries.

A negative factor for this component was the fact that the various technologies required to support this interface proved hard to integrate. This was because the technologies are quite new and the documentation available was limited. The solution to this problem required a complex redesign of the client component and meant the

scheduling of implementation had to be altered so that extra time could be spent solving the problem.

Overall, this interface was the correct choice of technology for the functionality required. The functions used by the interface allowed a simple and easy method to efficiently pass data between the server and client. Although a complex design was required the overall result proved the technology could be used to transfer data.

1.20.2 Sponsoring Company's View

A meeting was held with the General Manager of IDS to examine the system produced. A number of points were raised in the meeting. These points were broken into three categories and will be examined in following three sections.

5.1.1.25 Usability

The company believed that the user interface (UI) was too rough to focus on any specific type of user. It was commented that it would require customisation to refocus the elements onscreen and target the product towards a specific type of user. They also added that the reason the UI was not focused towards a specific type of user was because of the open ended specification that did not clarify who the target user was.

Besides the lack of the focus, the company was pleased with the interaction provided by the UI. It showed a number of features of the RIA language that IDS were looking for.

5.1.1.26 Further Development

The company has decided not to continue development on the system. The main reason behind this is the fact the company is currently re-developing a number of applications and does not have the resources to continue development. Although it will not be developed further, elements of this system will be helpful to add functionality to other products the company develops.

The most important aspect the company is interested in is the real-time data extraction and handling components in the system. The company plans to extract

these components and integrate them into the main reporting application they sell. They hope to add real-time data streaming as a new feature to the product to help interest clients with this requirement.

Another aspect the company is interested in is the 3D graph rendered on the Flex client. Although this feature did not function correctly it did prove that a 3D visualisation was possible and extra features like a movable axes could be implemented. This has sparked the company's interest in this type of visualisation and may lead to further development of it.

5.1.1.27 Fitness for purpose

The company required an investigation of the real-time elements of WITSML to determine the viability of moving into the real-time data market. The purpose of this project was to examine these real-time elements and implement a server around a number of requirements. In this respect, the system was a success. The system allows real-time data to be extracted from a WITSML server, cached in a database and viewed onscreen by a user. It has provided a detailed examination of the WITSML language and how it can be used with other components to create a system capable of real-time data streaming.

Conclusions

This section will discuss the conclusions discovered through the work performed in the project. It contains three sections each describing a different aspect of the project. Section 1.21 analyses the aims of the project and discusses if the produced system fulfils them. Section 1.22 contains a discussion on the work performed in the project and comments on the achievement of designing and implementing it. Section 1.23 concludes this section by discussing the idea of future development.

1.21 Analysis of Aims

The main aim of this project was to investigate the real-time data elements in WITSML. Through the construction of a web-based system, a large amount of information has been learned about real-time data and specifically, the way WITSML handles it. The system has shown that WITSML data can be extracted in real-time and passed efficiently through a series of technologies to display it on a web-based client. The system has also produced a keen insight into the technologies required to handle real-time data. The production of the system has allowed the key elements of real-time data streaming in WITSML to be discovered and has prototyped the idea of capturing this data. It has successfully investigated the real-time data elements in WITSML and has given the sponsoring company a glimpse into the possibilities for further development.

The secondary aims of this project were to investigate the viability of Adobe Flex and the JBoss application server as components in a web-based system.

- Adobe Flex was discovered to be a powerful tool that provided helpful functionality to design and create an RIA. It allowed interaction with a web-based server and successfully rendered different types of content received from this server. Adobe Flex provided all the features required in the client component of this project and would be ideal to use in any further web-based projects.

- The JBoss application server provided numerous built-in features and additional components that helped development of a web-based application. The components included in JBoss provided a good basis for further development. JBoss allowed all the complex elements of the system to be implemented in an efficient and structured manner. It would be ideal to develop further web applications and provides much more functionality than a simple web-container.

1.22 Reflection on Achievement

This project has taken a significant amount of time to design and implement. This has partly been due to a lack of knowledge of the new technology but also due to the number of components used in the system. The scale of the project was discovered to be much larger than first believed and although it did not run over the deadline time, significant work was required to complete a prototype. This has made the overall achievement of producing a working version of the planned design a great accomplishment.

Through the implementation issues section and evaluation section, a number of components were identified that could have been improved. Although this would have increased the functions available in the system, it is believed that the multitude of features provided make up for this incomplete functionality. Overall, it is believed an accurate prototype representation of the system has been created.

1.23 Future Work

If further development was performed on this system a number of elements would be improved. These improvements are listed below.

- *Core Library* – The system was developed so that it was easy to add further WITSML objects mapping and functionality to the core business logic. If development continued the number of supported WITSML objects would be increased. This would be achieved by developing new processing Java files, each specific to a WITSML object and adding these files to the configuration of the server. These files would simply inherit from the same Java interfaces created for the original project reducing the amount of work required. For these

features to be enabled on the client, the client would be updated to allow the selection of the new objects. Extra visual components may be needed to display these objects and would have to be added to the UI.

- *3D Graph* – The 3D graph currently used in the project is severely limited by the implementation used. If the system was developed further a different implementation of a 3D graph would be used and the features described in this report would be added. This would produce a fully interactive graph with a much richer display.

References

Borck, James. 2006. Adobe Flex 2.0 enriches the RIA development experience. *InfoWorld*. [Online] 10 August 2006. [Cited: 18 January 2008.]

http://www.infoworld.com/article/06/08/10/33TCflex_1.html.

Energistics. 2003. WITSML Data Schema. *WITSML Homepage*. [Online] 29 January 2003. [Cited: 20 November 2007.] <http://witsml.org>.

JBoss, Red Hat. 2007. JBoss Profiler. *Red Hat JBoss*. [Online] 2007. [Cited: 3 March 2007.] <http://wiki.jboss.org/wiki/JBossProfiler>.

Morearty, Mike. 2007. Common E4X Pitfalls. *Morearty Blog*. [Online] 13 March 2007. [Cited: 19 January 2008.] <http://www.morearty.com/blog/2007/03/13/common-e4x-pitfalls/>.

Morejon, Mario. 2006. A Look Under The Hood At Adobe Flex 2. *Channel Web*. [Online] 29 June 2006. [Cited: 18 January 2008.] <http://www.crn.com/it-channel/189700025>.

Ogbuji, Uche. 2006. Remove sensitive content from your XML samples with XSLT. *IBM*. [Online] 11 April 2006. [Cited: 18 November 2007.] <http://www.ibm.com/developerworks/xml/library/x-tipsensitive.html>.

Ort, Ed and Mehta, Bhakti. 2003. Java Architecture for XML Binding. *Sun Developer Network*. [Online] March 2003. [Cited: 30 November 2007.] <http://java.sun.com/developer/technicalArticles/WebServices/jaxb>.

OutwardMotion. 2007. JMS with JBoss. *Outward Motion*. [Online] 2007. [Cited: 12 November 2007.] <http://www.outwardmotion.com/outwardmotion/jmsjboss1.php>.

RedHat. Unknown. Hibernate EntityManager. *Hibernate Wiki*. [Online] Unknown. [Cited: 16 November 2007.] http://www.hibernate.org/hib_docs/entitymanager/reference/en/html_single/#configuration.

Saxena, Nihit. 2006. 3D Chart in Flex. *blog.myspace.com*. [Online] 21 July 2006. [Cited: 6 December 2007.] <http://blog.myspace.com/index.cfm?fuseaction=blog.view&friendID=13603991&blogID=147238737>.

W3C. 2004. Web Services Architecture. *W3C*. [Online] 11 February 2004. [Cited: 05 March 2008.] <http://www.w3.org/TR/ws-arch/>.