Project Number: MXC L072

ROSA II Web Service Prototype

A Major Qualifying Project

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Christopher Songer

Shaun Tyman

Date: October 11, 2007

Approved:

Professor Michael J. Ciaraldi, Major Advisor

Disclaimer: **This work is sponsored by the Missile Defense Agency under Air Force Contract FA8721-05-C-0002 (1 Apr 05 – 31 Mar 2010). Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.**

Abstract

With MIT Lincoln Laboratory's recent push in net-centric technology, many projects are looking to incorporate Web services. Radar Open Systems Architecture II (ROSA II), a sensor system framework, is no exception. This project designed and implemented a Web service prototype for use with ROSA II. The Web service allows for the creation and execution of a ROSA II system remotely. A Graphical User Interface, also created in the project, facilitates the process of creating the system.

Executive Summary

The purpose of this project was to design a Web service prototype that would provide access to the available components in a ROSA II installation. A user would then use a client in order to be able to design a system consisting of processing chains of these components and submit this design to the comtroller, or component controller, for execution. We determined that there were three main objectives that needed to be accomplished in this project:

- The creation of a new ROSA II component, called the "comjector", or component injector, which would receive the necessary information to design a valid ROSA II system and then reinitialize the comtroller with this new system description.
- The development of a Web service that would take a user-designed ROSA II system and submit a request to the comjector for starting up the corresponding components. The Web service would also be involved in retrieving a list of the available components in a given ROSA II installation and provide necessary descriptions for these components.
- The implementation of a client prototype with a Graphical User Interface (GUI) that would allow a user to easily design a ROSA II system and submit this request to the Web service.

Before we started to work on the three aforementioned objectives, we first researched the available technologies that were relevant to this project in order to determine their usefulness. We decided to use Apache Tomcat as the web server on which our Web service would be deployed, while Apache Axis2 served as the framework. Java was the programming language of choice in designing the Web service and client, as well as the GUI, which was done using the Swing Toolkit.

The comjector was created using a component generator tool provided in ROSA II. The necessary functionality that would unencode a ROSA II system description and publish a control parameter that would reinitalize the comtroller was added. The comtroller was then modified such that it would subscribe to this control parameter and start up one or more processing chains of components based on this new system description.

The Web service was implemented using Axis2 Data Binding with three operations defined in a WSDL. The "getComponents" operation was responsible for retrieving a list of interface descriptions corresponding with the available components of a ROSA II installation. The "submit" operation was used to take in the necessary parameters for creating a system description and sending this information in the form of an XML string to the comjector. The "getStatus" operation was used to determine whether the HTTP server corresponding with a particular comjector was currently running, which was useful prior to sending a request for submitting a system.

Finally, a client and corresponding GUI prototype was developed using Java's Swing Toolkit. Using the functionality provided by the Web service, the client allows a user to create a ROSA II system description by providing the necessary information in the appropriate data fields. Once a system is designed, the client invokes the submit operation in the Web service to run the system.

Acknowledgements

The project team would like to thank MIT Lincoln Laboratory for sponsoring this project and providing an exciting work environment and a challenging project. Without the continued support of the Laboratory, opportunities like the one provided to us would not continue. We would also like to thank Group 33, Ranges and Test Beds, for their support throughout the time we spent there. In particular, we would like to thank David Seth Hunter and Peter Jurgensen for technical advice and professional supervision for the project. Other thanks go to the Group 33 leaders, Gary Ahlgren, Frank Robey, Kurt Schwan, and John Nelson, who created the opportunity within the group by requesting WPI undergraduates for the project. Finally, we would like to thank Professor Michael Ciaraldi for advising the project and Emily Anesta for organizing the MQP program at MIT Lincoln Laboratory.

Table of Contents

Table of Figures

1 Introduction

Recently a large part of the computing community has been designing ‘net-centric’ software. This refers to participating as part of a continuously-evolving, complex community of people, devices, information and services interconnected by a communications network to optimize resource management and provide superior information on events and conditions needed to empower decision makers. One way to do this is by using a service-oriented architecture (SOA), and more specifically, Web services. Web services allow the invocation of functions remotely using communication over a network.

MIT Lincoln Laboratory (MIT/LL) is no exception in regards to the shift towards net-centric programming. The Laboratory has multiple projects in development that either focus on or involve a service-oriented architecture. Specifically, all three Worcester Polytechnic Institute (WPI) Major Qualifying Projects (MQP) for the year 2007 deal with Web services in some way.

A project currently in development at MIT/LL in Group 33 (Ranges and Test Beds), Radar Open Systems Architecture II (ROSA II), is also seeking a use for Web services. ROSA II is a sensor backend that includes both hardware and software for communicating with radars. Since ROSA II is more modular than its monolithic predecessor, it would benefit greatly from using Web services to both manage and run sensor systems. The use of Web services would also allow a user of a remote ROSA II installation to easily execute programs that are located off of the local network.

This project aimed to prototype a Web service that would serve as a discovery service and a management tool for a ROSA II installation. The Web service would allow the creation of an interface that discovers and presents to a user available components and resources. A client using this Web service would then allow a user to compose a system from available components and submit the design to be executed in the ROSA II installation. While this project is more of a proof of concept than a fully functional Web service, we have created three concrete applications used in the Web service:

- A new ROSA II component, the comjector, which handles communication between other components and the Web service.
- A Web service that supplies a directory service for available components/resources in a ROSA II installation. The service also takes a user-designed system and submits it to the aforementioned comjector.
- A client that communicates with the Web service and lets a user design a system based on available components. A graphical user interface was implemented in order to simplify this process.

By successfully completing these three main objectives, a more robust version of a Web service could be implemented by the developers of ROSA II in the near future.

2 Background

This chapter will provide an overview of the background research and experimentation that was performed as a precursor to the design and eventual implementation of the ROSA II Web Service Prototype. A number of technologies and concepts loosely related to those ultimately chosen are presented alongside those utilized to demonstrate a solid understanding of the state of the respective fields prior to the design decisions of this project being made.

2.1 Web Services[1]

The World Wide Web Consortium (W3C) defines a Web service as "a software system designed to support interoperable Machine to Machine interaction over a network". A Web service is essentially an API that can be accessed over a network and executed on a remote system. The most common styles of use for Web services are RPC, SOA and Representational State Transfer (REST).

- RPC Web services present a distributed function (or method) call interface that is familiar to many developers. Typically, the basic unit of RPC Web services is the WSDL operation.
- Web services can also be used to implement an architecture according to Service-oriented architecture (SOA) concepts, where the basic unit of communication is a message, rather than an operation.

[1] "World Wide Web Consortium". http://www.w3.org/2002/ws/. 2007.

- REST Web services attempt to emulate HTTP and similar protocols by constraining the interface to a set of well-known, standard operations (e.g., GET, PUT, DELETE).

While there is no one single specification for Web services, there are three "core" specifications that are commonly included in Web services: SOAP, WSDL, and UDDI.

2.1.1 SOAP[2]

SOAP is a protocol for exchanging XML-based messages over computer networks, normally using HTTP. SOAP may also be used over HTTPS in either simple or mutual authentication. SOAP was originally an acronym for Simple Object Access Protocol, but has more recently come to be used as an acronym for Service Oriented Architecture Protocol. SOAP details precisely how an HTTP header and XML file should be encoded in order for a program running on a particular computer to call a program running on another computer and pass information to it, as well as how the called program is permitted to return a response.

SOAP evolved from XML-RPC as additional functionality was added. The SOAP specification was originally designed with Microsoft's backing, but is now maintained by the XML Protocol Working Group of the W3C.

The following is an example of the formatting of a SOAP message requesting product information from a fictional warehouse Web service, in which the client wants to know which product has the ID 827635:

[2] "XML Protocol Working Group". http://www.w3.org/2000/xp/Group/. 2007.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetails xmlns="http://warehouse.example.com/ws">
      <productID>827635</productID>
    </getProductDetails>
  </soap:Body>
</soap:Envelope>
```

(“Wikipedia SOAP” http://en.wikipedia.org/wiki/SOAP 2007)

Figure 1 - SOAP request example

The following is an example of a possible response to the above client request:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetailsResponse xmlns="http://warehouse.example.com/ws">
      <getProductDetailsResult>
        <productName>Toptimate 3-Piece Set</productName>
        <productID>827635</productID>
        <description>3-Piece luggage set. Black Polyester.</description>
        <price currency="NIS">96.50</price>
        <inStock>true</inStock>
      </getProductDetailsResult>
    </getProductDetailsResponse>
  </soap:Body>
</soap:Envelope>
```

(“Wikipedia SOAP” http://en.wikipedia.org/wiki/SOAP 2007)

Figure 2 - SOAP response example

2.1.2 Web Services Description Language[3]

Web Services Description Language (WSDL) is an XML-based language that provides a model for describing Web services. It is essentially a description for how to communicate using Web services, defined as collections of network endpoints. WSDL is often used in combination with SOAP and XML Schemas to provide Web services over the Internet. A client program connecting to a Web service can read the WSDL to determine what functions are available on the server. Any special data types used are embedded in the WSDL file in the form of an XML Schema. The client can then use SOAP to actually call one of the functions listed in the WSDL.

WSDL documents traditionally define the following aspects of communications:

- Types – data definitions, as with other languages. Uses XSD traditionally.
- Messages – Messages are the basic unit of communication used by services. A message is defined as a series of parts, which may themselves be XML documents as defined by XSD.
- Operations – Operations define the actions performed by services (one action is one operation). Depending on the desired functionality, there are four kinds of operations: one-way, request-response, solicit-response and notification. In one-way operations, the service "can receive a message but will not return a response." In request-response, the service "can receive a request and will return a response." In solicit-response, the service "can send a request and will wait for a response." Lastly, in notification, the service "can send a message but will not

[3] "Web Services Description Working Group". http://www.w3.org/2002/ws/desc/. 2007.

wait for a response."[4] Messages in an operation are determined from the operation type, and are defined in the following table:

Operation Type	**First Message**	**Second Message**
One-Way	Input	n/a
Request-Response	Input	Output
Solicit-Response	Output	Input
Notification	Output	n/a

- Port Types – PortTypes list the sets of operations supported by an endpoint. This effectively describes what tasks the service can perform, and by nature of the operations defining messages, how it communicates.
- Bindings – Bindings describe protocols and data type definition for a port type. This is the section of the document where SOAP interfacing is handled (a SOAP operation would constitute the entire body of the bindings element).
- Ports – Endpoints defined by a binding and an address.
- Services – Collections of related endpoints.

The current WSDL specification has not yet been endorsed by the W3C, but it is currently up for consideration on the W3C website. WSDL exists as a mix of older similar purpose protocols, and has been created with input from several major computer-related companies.

[4] "W3Schools WSDL". http://w3schools.com/wsdl/default.asp. 2007.

2.1.3 Web Service Frameworks

There are a large variety of Web service frameworks currently available that facilitate the creation of both Web services and clients. Listed below are three of the larger ones that our group investigated prior to the project.

- **Apache Axis2**[5] - The Apache Axis2 project is a Java-based implementation of both the client and server sides of Web services. Axis2 is the spiritual successor of Apache Axis1.0 with Axis2 providing a complete object model and a modular architecture that makes it easy to add functionality and support for new Web services-related specifications and recommendations.
- **XFire**[6] - Codehaus XFire is a Java framework for development and consumption of Web services. XFire uses StAX (Streaming API for XML) for XML processing, resulting in better performance.
- **Apache CXF**[7] - Apache CXF is an open source services framework resulting from a merger between Xfire and Celtix. CXF is useful for building and developing services using frontend programming APIs, like JAX-WS. As of this project CXF was still in heavy development.

2.1.4 Apache Tomcat[8]

A Web service needs a web server to run on. Apache Tomcat is a web container/application server developed at the Apache Software Foundation (ASF). Tomcat implements the servlet and the JavaServer Pages (JSP) specifications from Sun

[5] "Apache Axis2". http://ws.apache.org/axis2/. 2007.
[6] "Codehaus XFire". xfire.codehaus.org/. 2007.
[7] "Apache CXF". incubator.apache.org/cxf/. 2007.
[8] "Apahce Tomcat". tomcat.apache.org/. 2007.

Microsystems, providing an environment for Java code to run in cooperation with a web server. Tomcat includes its own internal HTTP server.

2.2 ROSA II

Radar Open Systems Architecture II (ROSA II) is the next iteration of ROSA, and will become the new radar signal and data processing framework. ROSA II must add more flexibility, scalability, modularity, portability, and maintainability than is offered by the original version of ROSA. ROSA II will focus on enhancing these features, as well as providing a robust infrastructure for sidecar and other test bed development. A key enabler for this is abstraction, where interfaces among components are separated and defined. With abstraction in the software codes, new developers will not need to understand or modify complexities in the core code in order to add features or make changes for a new application. This should decrease the amount of code that needs to be maintained and increase the integration of code among applications. (From the ROSA II Vision Statement). The ROSA II goal is pluggable, modular software components for efficient, streamlined customization to specific applications. Major features of ROSA II include:

- Instead of the standard monolithic framework in which every part depends on a multitude of others, ROSA II aims to be more compartmentalized, modular, and layered.
- Instead of assuming intimate cross-function knowledge, functions in ROSA II are isolated through well defined interfaces. An example of this is details of SQL

being isolated in object implementation, not application. Essentially, data storage changes don’t affect application interface.

- In ROSA functionality was not always isolated. For example, the ROSA Integration and Detection module also generates display data. In ROSA II function is atomically isolated in modules, thus making it easy to substitute modules that conform to interface definition.
- ROSA II uses a pluggable framework so that instead of needing to do an entire system rebuild, local enhancement only requires rebuilding of relevant modules. This also allows the use of builds where only needed modules are built.

The ROSA II system runs via a series of components (sensors, data analyzers, displays) connected together in chains. For example, a component that generates pulse data, such as a sensor, will send that data to another component that takes pulse data as input. That component in turn will output a result in some format and can send this data to a component that takes as input the same data format.

3 Methodology

This chapter outlines the original plan proposed at the start of the project. Also included are the administrative details of how the project was organized, what tools utilized, and the general topics of where, when, and by whom the work was done.

3.1 Proposed Project Plan

The original plan for this project was broken up into four main steps: ROSA II introduction, new ROSA II component, Web service, and client.

3.1.1 ROSA II Introduction

The purpose of this first step was to become acquainted with ROSA II. This included obtaining, compiling, and running the latest copy of ROSA II. The main objectives of this step were to learn:

- How the component controller handles other components
- How to send and receive system parameters
- How to create new components
- How to access components over HTTP
- How, if possible, to determine the format of input/output data for a given component

3.1.2 New ROSA II Component

The second step was to create a new ROSA II component, the comjector, that would handle communication between the Web service and the component controller (comtroller). The component would accept system description files via HTTP. These files are in XML and tell the component controller what components to setup in a given system. The comjector would also handle status checking of the system and relate it back to the Web service.

The comjector was created via the component generation 'comgen' tool. This tool creates a component skeleton consisting of C++ source files for a new component with all of the necessary functionality. Components created in this fashion also come with their own HTTP server that can be accessed via the web. We used this server to pass messages between the component and the Web service.

Once we had the skeleton framework we modified the HTTP server code to accept an XML string that would be formatted as a system file. The comjector would then turn around and pass system parameters, a method of communication between components, to the component controller. The component controller would then start up a new processing chain or chains to run the user designed system.

3.1.3 Web Service

The third step was to create the Web service itself. First of all, we needed to decide which Web service framework would be most suitable for this project. Next we needed to decide which operations the service would provide. Initially we determined that the service would have three operations:

- The first operation would query ROSA II for available components/resources to supply to the user.
- The second operation would take in a large XML string formatted as a system description file and submit it to the comjector.
- The third operation would simply return the status of either the component controller or the user's system.

At the time of this project ROSA II had no easy way to return a list of components and the associated inputs and outputs of each one. It was suggested that we store this data in some sort of database as part of the Web service. While this is not a final solution, it would work for our project which is a proof of concept prototype.

The second operation would be accomplished as described above by sending the XML string over HTTP to the comjector in a RESTful manner. The third operation would ideally get the status of all running components from the comjector.

3.1.4 Client

The final step was to create a client that would interact with the Web service. Initially we used a text-based command line client for testing purposes. This provided us continuous feedback as to the progress of the Web service. The end goal, however, was to create a graphical user interface, perhaps by using the Java Swing GUI Toolkit. The user would invoke the first Web service operation, getting the list of available ROSA II components, and proceed to create a valid system via drag-and-drop, drop-down boxes, or some other implementation. The user would then submit this request back to the Web service. Before submitting the system description to the Web service, the client would invoke the status operation of the Web service.

3.2 Work Environment

The work for this project was performed at the MIT Lincoln Laboratory in Lexington, MA in Group 33, Ranges and Test Beds. The students worked full time on the project between August 23rd and October 11th of 2007, through Worcester Polytechnic Institute's (WPI) Global Perspectives Program in fulfillment of their Major Qualifying Project (MQP).

3.2.1 Integrated Development Environment

To facilitate rapid development in a shared environment, the team chose to utilize the Integrated Development Environment (IDE) Eclipse[9]. The Eclipse project is an open-source, open-architecture environment designed to provide a universal tool platform for software engineering. Specific functionality of the IDE is implemented through plug-ins, the most prominent of which is the Java Development Tools (JDT) package. By design, however, Eclipse is not limited just to Java development; via the use of other similar plug-ins, it can provide support for many other software engineering architectures and languages, such as C/C++, XML, and CORBA.

[9] http://www.eclipse.org

Eclipse provided the project team with numerous important benefits. First and foremost, the Java Editor aided in rapid development by providing code completion, error checking and resolution, syntax highlighting, and source code management. This resulted in less time being consumed by routinely poring over extensive Java documentation and manual recompilation of the source code. Eclipse's compatibility with the build system described in the next subsection provided easily accessible tools that were helpful for building the multitude of components inherent in a Web service system.

3.2.2 Build System

In addition to the build system provided by Eclipse, the development team created and maintained a standards-based build system for use with the Apache Ant[10] tool. Ant is designed to be a universal build tool reminiscent of, but greatly improved upon, the *make* utility. It has become the standard build tool for Java applications in many developer circles, and was seen as beneficial to this project by allowing a user to compile the source code easily without the need for installing the exceptionally-large Eclipse platform. Ant utilizes a hierarchical structure of XML files which define dependencies and tasks necessary for the process of building and running Java projects.

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

3.2.3 Version Control

One of the most challenging problems faced by teams of software engineers is the issue of concurrently modifying a shared set of files. The most commonly accepted solution to this is some implementation of a version control system, which not only provides a central repository that all team members can easily share, but also maintains a history of all committed revisions to each file. Additional features that are often provided by such systems allow developers to easily merge changes between file revisions, create branches of the main project tree, and automatically tag files with certain repository-related information.

For this project, the authors utilized the Subversion[11] version control system, which is designed to be a modern replacement for the older Concurrent Versions System. It provides all of the traditional features of CVS, with a number of new capabilities such as easier branching and binary file support. All source code was maintained through the team's Subversion repository.

3.3 Division of Labor

This project was a joint effort between both team members, Christopher Songer and Shaun Tyman. While both members worked in equal amounts on the project, the majority of certain sections were done by one person in particular.

[11] http://subversion.tigris.org

The methods of communication between the 'comjector' and the 'comtroller' were mainly developed by Christopher. He also did the majority of the Graphical User Interface (GUI) work because of his familiarity with Swing.

Shaun worked mainly on the Web service and the communication that passed between it and the client and comjector. He also created the component list and all the associated code to facilitate the listing of components.

The paper was done equally between the two members, with each person writing the sections that they worked most closely with throughout the project.

3.4 Procedural Timeline

Week One

- Project proposal
- Proposal presentation
- Introduction to ROSA II Components
- Creation of "Comjector" skeleton
- Preliminary implementation of Web services
- Rudimentary communication established between Web service and "Comjector"
- Makeshift command-line/graphical client interface created for testing purposes

Week Two

- Completed implementation of "Comjector" component
- Implemented communication between "Comjector" and "Comtroller"
- Started to create additional Web service operations that allow for retrieval of available components

- Started to implement functionality of client and GUI
- Wrote rough draft of Introduction/Background information for report

Week Three

- Finalize component data structure retrieval from Web service
- Completed a first revision of a functional client and GUI
- Continued writing report, including Methodology section

Week Four

- Continued writing/revising report, specifically Results and Analysis
- Added improved XML parsing implementation in Comtroller
- Improved functionality of Client/GUI

Week Five

- Heavy report writing
- Finished Client GUI
- Began presentation work

Week Six

- Rough Draft of entire report
- Rough Draft of presentation
- Finished testing Web service

Week Seven

- Final Presentation
- Completed project

4 Design and Implementation

This chapter presents a description of the design requirements for the project as well as the final design of the ROSA II Web Service Prototype. A detailed discussion of the implementation of each subsystem by its respective author is also included.

4.1 Requirements

- Create a net-centric/SOA/Web-service interface
- Discover and present to a user the components and resources available in a ROSA II installation
- Allow the user to compose a system from the components
- Submit the design back to the component controller, which would then execute the system based on the user's design
- Conformity with the ROSA II system file format.

4.2 Overview

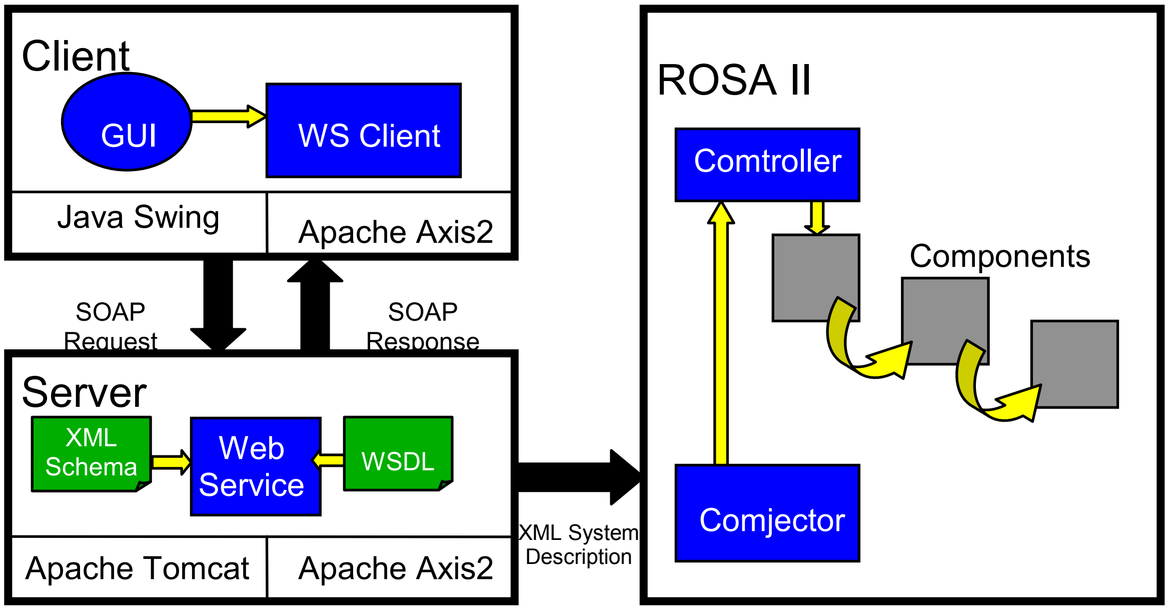

Figure 3 - Design Overview

Figure 3 shows the general layout of how our project fits in with ROSA II and all the communication channels involved.

As stated in the previous section, a lot of our system design decisions were based off of the current format of ROSA II system files, which can be seen in Figure 4.

```
<?xml version = "1.0" ?>
<ROSA2_system >

 <metaData
  title = "HAYSTACK DEMO A"
  name = "Demo A"
  location = "Lexington"
  exePath = "/home/dc/workspace/rosa2_U/bin"
  RTCLconf = "/home/dc/workspace/rosa2_U/bin/rtcl_commcfg.xml"
 />

 <cans>
  <can name = "rosa2-hay"/>
```

```
</cans>


<chains>
  <chain name="common" instances="1" description="components common to system"/>
  <chain name="main" instances="3"  description="the MAIN chain"/>
  <chain name="test" instances="3"  description="an experimental TEST chain"/>
</chains>


<components>
  <component chain="main" runOnCan = "1" canSpan = "1"
          startupCmd = "noncohint 10 2 PulseAuxStream/#S
          local_NonCohInt/#S -componentID=noncohint_#S --rtclconf "/>


  <component chain="main" runOnCan = "1" canSpan = "1"
          startupCmd = "davedetect -componentID=davedetect_#S
          --inputTopic=local_NonCohInt/#S --outputTopic=mttDetectionRecord
          --controlTopic=davedetectControl/#S --rtclconf"/>


  <component chain="common" runOnCan = "1" canSpan = "1"
          startupCmd = "track_file_manager -componentID=track_file_manager
          mttStateVector  --rtclconf"/>


  <component chain="main" runOnCan = "1" canSpan = "1"
                    startupCmd = "generate_uam PulseAuxStream/#S #++ sum eth4
          -componentID=generate_uam_Raw_#S --rtclconf"/>


  <component chain="main" runOnCan = "1" canSpan = "1"
          startupCmd = "generate_uam local_NonCohInt/#S #++ sum eth4
          -componentID=generate_uam_Int_#S --rtclconf"/>


</components>


</ROSA2_system >
```

Figure 4 - Example ROSA II system file

4.3 Details

This subsection outlines the issues faced and gives implementation details involved in the project.

4.3.1 Comjector

Instead of writing our component from scratch, we took advantage of the comgen tool which is included in any ROSA II system. Comgen is essentially a script that, after prompting for the name of the new component and a few other parameters, will create a fully functional component. Although the component did not actually do anything, all the necessary classes and files had been created.

The first step was to strip out code that dealt with input and output of the component, which was not going to be used. Next we tested out the HTTP server and familiarized ourselves with how to run the component. The included HTTP server came with a page that accepted input from a form and printed it back to the user on the Web page. We created a new function modeled after this called runComtroller_cgi and registered it to the page runComtroller.cgi. The method was modified to accept arguments in a RESTful format and on being called it would create a properly formatted system file.

Next we needed a way to communicate with the comtroller to tell it to run on the new system file. By using more of the built-in functionality the comjector was able to publish a control parameter, aimed at the comtroller, that specified the location of the system file the comjector made. We then modified the comtroller to recognize this control parameter message and re-initialize itself with the given system file.

We then tested the communication between the two components by accessing the comjector's runComtroller.cgi page with different arguments and making sure the comtroller was restarting and running the right components.

Later on we took the system file creation logic out of the comjector and put it in the Web service. This was done because the role of the comjector is to facilitate communication between the Web service and ROSA II; the Web service should be the only one doing work. The comjector still accepts arguments through HTTP, but now it looks for a header, creates a file, and continues to add to this file until it receives a footer, at which point it completes the file (including setting the appropriate local variables) and passes its location on to the comtroller.

4.3.2 Web Service

Prior to the official start of this project we had been working as interns at MIT/LL. During this time we experimented with three different Web service frameworks: Xfire, CXF, and Apache Axis2. After some deliberation and the helpful assistance of Gary Schorer, another MQP student who had worked more closely with theses frameworks, we chose Axis2 as our Web service development platform with Apache Tomcat as our web server.

After initial experimentation with the various methods of Web service creation included in Axis2, we decided on a POJO (Plain Old Java Object) service deployment. We created our service as a normal Java class and place it along with the associated services.xml shown in Figure 5 into Axis2's Web services directory as an '.aar' file. Axis2 handles the deployment of the service and creates a WSDL to describe it.

```
<!--
  ~ Licensed to the Apache Software Foundation (ASF) under one
  ~ or more contributor license agreements. See the NOTICE file
  ~ distributed with this work for additional information
  ~ regarding copyright ownership. The ASF licenses this file
  ~ to you under the Apache License, Version 2.0 (the
  ~ "License"); you may not use this file except in compliance
  ~ with the License. You may obtain a copy of the License at
  ~
  ~ http://www.apache.org/licenses/LICENSE-2.0
  ~
  ~ Unless required by applicable law or agreed to in writing,
  ~ software distributed under the License is distributed on an
  ~ "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
  ~ KIND, either express or implied. See the License for the
  ~ specific language governing permissions and limitations
  ~ under the License.
  -->
  <service name="ROSA2Service" scope="application">
    <description>
       ROSA2 POJO Service
    </description>
    <messageReceivers>
       <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-only"
                  class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver"/>
       <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
                  class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>
    </messageReceivers>
    <parameter name="ServiceClass">rosa2.service.ROSA2Service</parameter>
</service>
```

Figure 5 - Services.xml file

We constructed an ant build file to automate everything from compiling, packaging, and copying of the service into tomcat. Also included in the build file were the steps necessary to create a client from the WSDL.

In order to test functionality of the Web service we initially created three simple operations for the Web service: submit, getComponents, and getStatus. The getStatus operation simply returned the status code of the comjector's HTTP server. GetComponents only returned an array of dummy strings. Submit took in several strings and string arrays containing ROSA II system information, formatted them into the appropriate URL, and then called GET to trigger the comjector. While these operations were by no means final, they provided a working Web service to build off of, and assured us that clients could be created from the associated WSDL.

The changes and enhancements made to each operation throughout the project are explained in detail below:

<u>**GetStatus**</u>

Unfortunately we did not have sufficient time to improve upon this operation. In its final form, the user gives the Web service a hostname and port, and it will return a HTTP status code indicating whether or not there is a web server currently running on that address.

<u>**Submit**</u>

The submit operation originally took in arguments for name, title, location (typed as Strings) and lists of machines, chains, and components (formatted as String arrays). The service took these arguments and formed them into a correctly formatted URL with the arguments placed in a RESTful manner (argument=value&argument=value&...), and submitted it to the comjector.

The first modification to the operation consisted of including the hostname and port number of the machine running the comjector component. This was done to allow the client to specify which ROSA II system it wanted to access in the event of multiple installations on one machine. This also had the effect of making the Web service independent of a given ROSA II installation.

The second major change to the submit operation was creating new classes for chains and components. As previously stated, in order to get the service working the chains and components were originally represented as arrays of Strings, with each component string having all the necessary information separated by colons. Once we were experienced with the functionality of the Web service we created these following new classes: Chain and ComponentCommand (different from the Component class that the service returns with getComponents()). These classes hold all the information necessary to create a chain or component inside a ROSA II system file.

The current method signature for the submit operation is as follows:

```
submit(String hostname,
       int port, String title,
       String name,  String location,
       String[] machines,
       Chain[] processChains,
       ComponentCommand[] components)
```

GetComponents

At the time of this project, ROSA II, being heavily in development, did not have the capability to list installed components and their input/output data formats. In order to get around this, we decided to create a local directory of components in the service itself. The components were listed in an XML file in the format specified by the schema shown in Figure 6.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="componentList">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="component" minOccurs="0" maxOccurs="unbounded"/>
        <xs:complexType>
            <xs:sequence>
              <xs:element name="argument" type="arg" minOccurs="0" maxOccurs="unbounded"/>
              <xs:attribute name="name" type="xs:string" use="required"/>
              <xs:attribute name="path" type="xs:string" use="required"/>
              <xs:attribute name="description" type="xs:string" use="required"/>
            </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>


<xs:complexType name="arg">
   <xs:sequence>
            <xs:element name="argument" type="xs:string"/>
            <xs:attribute name="type" type="xs:string" use="required"/>
            <xs:attribute name="visible" type="xs:boolean" use="required"/>
            <xs:attribute name="required" type="xs:boolean" use="required"/>
            <xs:attribute name="format" type="xs:string" use="required"/>
            <xs:attribute name="text" type="xs:string" use="required"/>
   </xs:sequence>
</xs:complexType>

</xs:schema>
```

Figure 6 - Component List Schema.

As stated previously, the getComponents operation initially returned an array of strings. We quickly realized that we would need a custom Component object to hold all of the information included in a component such as name, description, input, output, and arguments.

At this point in time we had been using the JiBX data binding framework to create our clients. However, when running the WSDL2Java script, JiBX could not unwrap the Component object from the WSDL. We once again consulted Gary Schorer, another MQP student, who recommended we use Axis2 Databinding Framework (ADB) to create our clients instead. The conversion from JiBX to ADB was relatively easy, with the only changes appearing in how the client invokes methods.

In an effort to make the Web service independent of any given ROSA II installation we opted for having the getComponents operation take a hostname as input. This would prompt the Web service to look for the appropriate components list file to return the corresponding components. While our Web service only has one componentList and does not actually do anything special if given a specific hostname, the functionality is there if a future developer should want it. After finding a componentList, the service creates an array of Component objects which are passed back to the client.

In keeping with the modularity of Web services, even though the method returns the custom Component object, the client does not require knowledge of the source code. Axis2 defines the Component class and any other custom classes in the WSDL that describes the service.

4.3.3 Client

One of the other major requirements of this project was the creation of a client that would allow the user to submit requests to the Web service, which would in turn communicate with the comjector and pass the request to the comtroller for execution. The actual implementation of the client consists of a single Java class that implements methods based on the operations defined in the WSDL. By using the WSDL2Java script that is provided with the Axis2 installation, stub classes were created that would handle the communication between the Web service and the client. Using ADB data binding (previously using JiBX data binding, as explained in the previous subsection), invoking an operation on the service was as trivial as creating a new Java class that implemented the automatically created stub and calling the appropriate method using the stub. Thus, the creation and implementation of a working client was found to be quite simple.

In order for the client to be truly usable in a typical scenario involving the ROSA II Web service, a user interface needed to be implemented. While the Java class implementing the client itself only contained the logic needed to submit requests and retrieve responses from the Web service, the Java class that implemented the GUI provided all of the necessary functionality for allowing the user to both easily submit requests for a particular ROSA II system and receive status information related to various components in the ROSA II installation.

First Iteration

The first GUI that was designed for the client was far from being feature complete, in that it only consisted of a window with three buttons: one button for submitting a complete ROSA II system file, one button for retrieving a list of all available components in a particular ROSA II installation, and one button for retrieving the status of the comjector server. The sole purpose of this rudimentary user interface was to test the use of Swing in creating a front end for the client that would in turn communicate with the Web service. Once it was determined that this version of the GUI was implemented correctly, namely the information needed to construct a complete ROSA II system file was submitted successfully to the comjector, which then allowed the comtroller to run a system based on this configuration, it was deemed necessary to expand the functionality of the GUI such that a user could input data corresponding with a system file of his or her own creation.

Second Iteration

Through the use of Java and the Swing toolkit, the GUI was expanded such that it contained the necessary input fields for entering data that would ultimately become a properly formatted ROSA II system file. The first section of input fields that is part of the GUI is related to information needed for the Web service to interact with the comjector. Specifically, the Web service must know the hostname and port of the comjector's own web server in order to communicate with the comjector and its corresponding ROSA II installation. The next section of input fields in the GUI corresponds with the first section of the system file, the metadata. In this group of text fields, the user is able to enter the name, title, and location relevant to the particular ROSA II system that he or she is designing. Although the information entered here does not ultimately affect the actual execution of the ROSA II system, this information allows the user to give a brief amount of information about the newly created system file. Two attributes defined in the metadata of the system file, “exePath” and “rtclConf”, provide information about the location of the component executables in a given ROSA II installation and the location of the RTCL configuration file needed by the installation, respectively. This information is not provided by the user, but instead by the comjector that he or she designates through the given hostname and port, since it has been assumed that there would be only one ROSA II installation corresponding with each instance of the comjector. Therefore, the comjector is responsible for providing the system file with the locations of the component executables and RTCL configuration file prior to running the comtroller.

The remaining sections of input fields correspond with areas of the ROSA II system file that actually affect which components are to be run, how these components are arranged in chains, and on what computers these components are executed. The section of input fields on the left side of the GUI allows the user to enter the list of computers, or “cans”, on which he or she wants the desired components to be executed. By entering the name of a computer in the designated text field and clicking the appropriate button, the computer's name will now appear in the list of computers to be included in the system file. Should the user decide that he or she does not want one or more computers in this list to be included in the system file, he or she can select one or more computers from this list for removal and then select the button associated with deleting these entries from the list.

The section of the GUI that takes up the majority of the space is associated with the chains of components that make up the underlying system to be executed by the comtroller. When the client's GUI is first launched, this area appears to be mostly blank, except for a single button that allows the user to add a chain to the system. Once a chain is added, more input fields relevant to this newly created chain are added to the GUI, such as text fields that allow the user to enter the name, number of instances, and description of the chain. In addition, two new buttons appear: a button that allows the user to add a second chain to the system file and a button that allows the user to add a component for this first chain. By clicking on the first button described, the user is able to continue to add multiple chains in the system, which will in turn each allow the user to specify the relevant information pertaining to the chain. By clicking on the second button, a group of input fields relevant to a component in the chain appears below the general chain information. Using these text fields, the user is able to enter information regarding what computer(s) the component will be executed on and the component's startup command. Once this information is added, the user is then able to either add more components to this particular chain in a similar fashion as the previous component, add more chains to the system, or submit this system request to the Web service to be then sent to the comjector for execution by the comtroller.

The three buttons as described in the first implementation of the GUI are once again present in this feature-complete version of the interface. Now, the button responsible for submitting a complete ROSA II system file to the comjector uses the information provided by the user in the appropriate input fields in the GUI and sends a request to the service in order to construct an appropriate system file based on this data. In addition, the buttons that retrieve a list of available components in a given ROSA II installation and the status of the comjector use the hostname and port of the comjector's server provided in the GUI in order to retrieve the information relevant to the desired comjector.

Third Iteration

Just as this first fully functional GUI improved on the very basic interface designed at the project's inception, further improvements have been made to the GUI in order to increase functionality and ease of use. The functionality provided by the button responsible for retrieving a list of available components in a ROSA II installation is now used by combo boxes in the GUI in order to display a list of available components that can be selected. Instead of the single text area containing the startup command and arguments needed for a particular component to execute, the user is able to select one of the available components in the ROSA II installation to go into a particular chain. Once a component is chosen, various information pertaining to this component, such as text fields for each argument, combo boxes listing every component in the current chain that outputs a data type that is compatible with each of the necessary inputs, and labels listing this component's outputs are made visible. Using an included XML file, the service now is able to format each of the component's arguments correctly, provided by the corresponding input topic and other argument selections, and use this information in order to create a properly formatted ROSA II system file.

Another addition to the GUI which is not necessarily Web service-related, but may be implemented as a Web service in the near future, is the ability to save the current state of the ROSA II system currently being created. By pressing a button on the GUI, all information currently entered into the data fields, such as machine information and the number and types of components in the system, are stored in a text file located in the client’s directory. By pressing another button, all previously saved information is restored.

5 Results and Analysis

This section describes the implementation results, how all goals were met, and the finished deliverables. It also gives a walkthrough of setting up and running the Web service.

5.1 Implementation Results

The final version of the Web service system included all the original specifications along with others that were added as the project progressed. The overall design was well thought out and organized, to which we attribute the success of this project. Major roadblocks that could have been reached were discovered beforehand, allowing the implementation to follow smoothly.

The development of a new ROSA II component termed the 'comjector' was successfully completed. When accessed through its HTTP server with a simple HTTP GET command it created a system file and told the comtroller to re-initialize itself.

The Web service was also fully implemented and contains three operations. The submit operation takes in data pertaining to a ROSA II system, assembles it into a system file, and sends it to the comjector. The getComponents operation returns a list of components retrieved from a XML component list file. The status operation returns the status of the comjector in terms of whether it is running or not.

The client created to demo the Web service ended up being more complete than we originally thought. The client provides a graphical interface to creating a system file by allowing a user to compile a list of ROSA II components to run and include other relevant data.

5.2 End-to-End Example

Presented in this section is an end-to-end usage example of the entire ROSA II Web service prototype system. It is intended to provide the reader with a clearer understanding of exactly how the system was designed to be used in a real-world situation. In this example, the user will be running the client with the GUI that was created as part of the project. It should be noted, however, that a developer can design their own client and/or UI for the service based on the WSDL.

5.2.1 Set-up

In order for the Web service to be functional, there are certain prerequisites of which the operator must be aware.

<u>ROSA II</u>

- Create a new ROSA II installation or use an existing one.
- Deploy the comjector in the components folder.
- Start up an instance of the comtroller using the included default.system file or a system file of your own.
- Start up an instance of the comjector making sure to specify the arguments for http-port, exePath, and systemPath (respectively - port to run HTTP server on; path of ROSA II executables; path to store system files created).

Web service

- Download and install Tomcat or a web server of your choice.
- Install Axis2 in the web server.
- Create environment variable AXIS2_HOME and set to Axis2 base directory.
- Create environment variable TOMCAT_HOME and set to Tomcat base directory.
- Modify the componentList.xml to accurately describe components available in the relevant ROSA II installation.
- Run the ant build script: ant generate.service. This will compile the service and place it into Axis2's services directory, effectively deploying the service.
- Start up your web server (Tomcat).

5.2.2 Client

Creating the Client

Creating, compiling, and running the client is as simple as running our ant build script on the target 'client.run'*. Note that you will need to specify the URL of the Web service's WSDL (default is localhost:8080). This script runs Axis2's WSDL2Java script on the service's WSDL to create service stubs that handle communication between the service and the client. It then compiles and runs the client and GUI created in this project.

*Advanced users can create their own clients from the services WSDL.

Running the Client

Once the client has been executed, the user will be presented with a graphical user interface intended to make system file creation easier. First, the user will want to enter the basic data such as the location of the ROSA II installation they are trying to access. Next, the user determines the components that he or she wishes to include in the submitted system. The GUI calls the getComponents operation of the Web service in order to get the interface descriptions of the available components. The user can then arrange these components into processing chains as they see fit. Finally, the user submits the system to the Web service by calling the submit operation. At this point everything else is done behind the scenes, with the user only being able to check on the status of the comjector to see if it is still running.

5.2.3 Behind the Scenes

After getting data through the submit operation, the service converts it into a large XML string formatted as a ROSA II system file. The service then splits this up into a header, the main body, and a footer. This is done because if the system file is very large the service will need to send it in multiple segments, and the comjector will need to know the start and end of a file. At this point the service will HTTP encode the strings and send them over HTTP to the comjector.

When the comjector receives a new header through its web server it will create a new system file. The comjector will continue to append information to this file on subsequent HTTP calls until it receives a valid footer. At this point the comjector finalizes the file and, using ROSA II control parameters, sends a command to the comtroller telling it to re-initialize itself using the new system file. Finally, the comtroller runs the components with the arguments provided in this new system description.

6 Conclusions

From the start, our group carefully planned out our project by being careful not to overestimate how much we could accomplish in the short seven week period, while also guaranteeing a significant implementation that could adequately accomplish the project specification as defined by the group sponsors at MIT Lincoln Laboratory. We closely followed the software engineering process, by first creating the design, analyzing use cases and potential problems, and then starting the implementation phase. The actual development was done in iterations, allowing initial features to be perfected while introducing new features as the project evolved.

6.1 Outstanding Issues

Due to the limited amount of time we had to work on this project, many features were either not implemented or implemented as "hacks" due to time constraints or the limited scope of the project.

For instance, the list of available components is hard-coded in a configuration file on the Web service instead of being dynamically created by the comjector.

6.2 Future Work

There are some components that could be completed as future work to our project. It has already been stated, but it is important enough to iterate this again: our project is a fully functioning prototype that allows a user to remotely submit a system description to be executed by a comtroller in a ROSA II installation. However, due to time constraints, there are features that could either be added on or improved upon to both increase the usability and performance of the application.

First of all, the retrieval of the list of available components could be implemented as a true Web service. Currently, the list of components and their descriptions are stored in a configuration file in XML format on the server. However, if this function were to be truly useful, the Web service should be able to submit a request to the comjector for a list of all available components and their corresponding descriptions in a particular ROSA II installation. This list would be dynamically created and used by the client for displaying what components are available in the creation of a particular ROSA II system description.

Likewise, the Web service could include an operation that would create a list of all available machines on the network that can be used to run the various components in the processing chains of a ROSA II system. Instead of the user manually entering the names of one or more computers on the network to run the components, the user would be able to select from a dynamically created list of available machines at runtime.

Finally, the GUI could be improved in order to improve its functionality and intuitiveness. For example, the act of specifying a processing chain of components in a system could be represented as drawing arrows between component objects.

Appendix A: Sponsor Organization

From the MIT Lincoln Laboratory website:

> "MIT Lincoln Laboratory has pioneered in advanced electronics since its origin in 1951 as a Federally Funded Research and Development Center of the Massachusetts Institute of Technology.
>
> "The Laboratory's fundamental mission is to apply science and advanced technology to critical problems of national security.
>
> "The scope of the problems has broadened from the initial emphasis on air defense to include communications, space surveillance, missile defense, tactical surveillance systems, air traffic control as well as air defense. Throughout its history, the Laboratory has had an extensive program in advanced electronics technology which has led to major advances across the breadth of its programs.
>
> "MIT Lincoln Laboratory has maintained a consistent approach to problem solving. The approach emphasizes following a project from the concept stage, through simulation and analysis, to the development of hardware and the ultimate demonstration of an integrated system. The Laboratory's environment of dedicated people, in well-equipped state-of-the-art facilities, motivates excellence and innovation.
>
> "Programs involving new graduates and experienced professionals provide opportunities to work with experimental systems in the field, as well as to conduct laboratory investigations and theoretical studies, from fundamental component development to complex system design."