FreeSB User's Guide

1. Introduction

1.1 Purpose

This document provides step-by-step instructions on using FreeSB to write custom FreeSB services and to integrate applications using the FreeSB.

1.2 Background

FreeSB is an application integration framework written in Java, and based on the Enterprise Service Bus (ESB) concept. It enables loosely coupled services (applications) to plug-in to the bus and communicate with each other. The framework uses SOAP as the transport mechanism to communicate between these services, thus platform interoperability is not an issue. The Apache Axis SOAP implementation is used as the transport.

FreeSB comes with some pre-written services and components required for its operation:

- AuthService – This is the authentication & authorisation service that handles all security related matters within FreeSB
- EsbClient – This is the client module that applications use to "plug" into FreeSB
- EsbCommons - Contains all the libraries required by the FreeSB as well as code templates and build scripts
- EsbDocs – Contains the documentation for FreeSB
- EsbLogger – This handles all logging related activity within FreeSB and it's client applications
- SupervisorService – This is the management service for the FreeSB, it acts as both a web front-end in order configure the FreeSB, as well as a co-ordinator for FreeSB services to perform registering (or "plugging" in) and load-balancing/fail-over
- TestService – Features a sample web application that showcases functionality such as performing calls both synchronously and asynchronously

2. FreeSB Architecture

2.1 Introduction

<Insert Diagram here>

Note that when a "custom" service is referred to, it is an external service that has been written by a user.

Also note that names of services have been shortened to what they are configured in FreeSB with, i.e. instead of referring to Authentication Service, it is referred to as AuthService, as that is the name defined for that service.

Each service that is required to "plug" into FreeSB will require a corresponding client that can call the methods in that service. The EsbClient that is provided with FreeSB provides base functionality by acting as a client for the core FreeSB services such as the AuthService, SupervisorService and EsbLogger. However, if the user wishes to implement a custom service, then the EsbClient needs to be extended in order to handle the new service and it's methods. This new "custom" client would make use of its EsbClient super-class to perform the core tasks such as authentication, logging etc, but would have it's own logic to handle operations on the new custom service.

Every service also needs to have what's called an "adaptor" – a class or other such programmatic unit that would make and receive the actual FreeSB calls. This adaptor would call another service by using the corresponding custom EsbClient written for that service.

The calls that are made are serialized into SOAP and would arrive at the requested service over the HTTP protocol (note that HTTPS can be used as well, to encrypt the data that is sent). Here, there would be another "adapter" attached to that service which would intercept the call, and perform such actions as authentication and authorisation, before passing the call onto the service implementation itself, (be it another class or a mainframe etc) which would perform the requested task and return the result back to the caller.

If the task requested takes a large amount of time, it is possible for the called service to not block – instead it would return the call (without a result) immediately back to the calling service. The called service would continue and process the task *asynchronously*. This means that after the task is completed, the called service would have to send the result back to the original calling service, as long as the calling service's adaptor has been configured to handle incoming requests.

This User's Guide will focus on developing using the Java platform, as it is currently the only client supported by FreeSB. Although it is possible to use any other language or platform that supports SOAP to communicate with FreeSB services, a client will need to be written for each such platform. FreeSB uses the Apache Axis implementation of the SOAP protocol for communicating with internal services, and it is the SOAP implementation used for the examples presented here. To follow the examples in this guide, it is **strongly** recommended that the intricacies of Apache Axis be learnt before proceeding. Documentation for Apache Axis is available at the following URL:

http://ws.apache.org/axis/java/user-guide.html

Pay particular attention to the section on "Using WSDL with Axis".

For developing services and integrating applications outside of Java (for example, .NET) please refer to the appropriate documentation for the relevant platform.

This guide will first show how to write a service (including a corresponding adaptor) and make it accessible to the FreeSB. Then the guide will show how to write a client (that extends the EsbClient) to access this new service, finally showing how to use this custom client within an application to make calls on the new service (including implementing an asynchronous call-back mechanism). This will assume the reader has followed the instructions in the "FreeSB Installation Guide" and that an installed and functional FreeSB is available to use.

3. Writing A Custom Service

To write a custom service requires several steps, including creating the service implementation class as well the corresponding adaptor, finally configuring FreeSB to accept this new service.

3.1 Writing the Service Implementation Class

The service implementation contains the functionality that is to be accessed via FreeSB. It can be anything, a mainframe application, a database etc as long as the adaptor (which translates FreeSB calls) can access it. In this case it is a normal Java class, and hence its methods can do anything possible in Java, with a few limitations:

- Time – If the processing within a method takes an extraordinary amount of time to return (i.e. greater than a few seconds) then the method should be implemented as an asynchronous call i.e. it spawns off perhaps a new thread to do the processing when a request arrives, returning the call to the client immediately without a result. After processing, the thread can return the result to the client by using a "call-back" method (see later chapters on implementing asynchronous call-backs).
- Types – Whilst the methods themselves can do anything possible in Java, the input parameter type(s) and return type must be SOAP-compatible. Here is a list of compatible types:

  byte[]
  boolean
  byte
  java.util.Calendar
  java.math.BigDecimal
  double
  float
  byte[]
  int
  java.math.BigInteger
  long

javax.xml.namespace.QName
short
java.lang.String

For example, here is a simple “Hello World” Java class, which has a method that accepts a String called “name”, and returns a greeting with the name inserted:

HelloWorld.java

```
package com.spherion.ap.esb.helloworldservice;

public class HelloWorld
{
   public String getMessage(String name)
   {
      return “Hello” + name + “, this is world!”;
   }
}
```

Despite the trivial example, that is really all that is required to implement functionality in a service, a Java class that takes valid argument types and returns a valid return type. Note that in future, this could be a C# class, or any programming language – as long as the service can be accessed via SOAP, it’s compatible with FreeSB (with a valid client of course).

Next is the adaptor class, which is what is “exposed” to the outside world through SOAP, and accepts incoming requests, passing valid ones along to the service implementation. In this case no translation of requests into compatible formats is required, as both the adaptor and service implementation is written in Java but if the service implementation was a mainframe application, then the adaptor would need to translate the method calls into the proprietary format accepted by the mainframe application. For the example below, it is assumed the adaptor is in the same Java package as the service implementation:

HelloWorldAdaptor.java

```
package com.spherion.ap.esb.helloworldservice;

import com.spherion.ap.esb.client.EsbClient;
import com.spherion.ap.esb.client.EsbPrincipal;

public class HelloWorldAdaptor
{
   private HelloWorld hw;
   private EsbClient ec;

   public HelloWorldAdaptor ()
   {
```

```
        try
        {
            //instantiates the EsbClient and the HelloWorld class
            this.ec = new EsbClient("HelloWorld");
            this.hw = new HelloWorld();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }

    public HelloWorldAdaptor(com.spherion.ap.esb.helloworldservice.HelloWorld hw)
    {
        this.hw = hw;
    }

    public String getMessage(String esbSeqId, EsbPrincipal[] principals,
                             String name) throws Exception,HelloWorldException
    {
            //checks the security credentials of the requesting user and service
            if(ec.checkPrincipals(principals))
            {
                return hw.getMessage(name);
                //ec.sendResponse(esbSeqId, "Message successfully logged");
            }
            else
            //if it was invalid, throw an exceptiom back to the client
            {
                throw new HelloWorldException("One or more principals " +
                "is not valid!");
            }

    }
}
```

As can be seen, there's a little bit more work to do in writing the adaptor:

- Firstly, the EsbClient class needs to be imported into the adaptor
- Then, the EsbPrincipal class is imported. As described earlier in the description of the AuthService <<<Err?>>>> , a principal is a token that tracks a user or service's session within FreeSB, and includes information on the roles they belong to.
- Declare two class variables, one referencing the EsbClient and the other the service implementation class, which in this case is "HelloWorld"
- Declare a no-argument constructor for the class, which will instantiate the service implementation ("HelloWorld") and EsbClient objects. We pass in the name of the service as a String argument to the constructor of the EsbClient, to tell it that it will be acting on behalf of that service. FreeSB

will need to be configured to accept “HelloWorld” as a valid service, please see further below.

- Declare a constructor for the class that takes an instance of the service implementation (“HelloWorld”) object and store it in the appropriate class variable.
- Then, all that is needed is to have methods identical in name to that of the service implementation class, in this case getMessage(String name) - except for the fact the method in the adaptor accepts two more arguments (that could be best described as “header information”) at the start, the ESB Sequence Id which is a unique number for all messages sent using FreeSB, and an array of Principals, which contain the service and user principals. Another difference is that the methods in the adaptor should throw a custom exception for the service, in this case an instance of HelloWorldException
- Within the method, the principals that are associated with the request are checked for validity by using the EsbClient’s checkPrincipals() method. This will use the AuthService to check them, and if they are valid principals, then the request is passed on to the service implementation for processing, minus of course the “header” information. The result is returned back to the client.
- If the principals are not valid, then a HelloWorldException is thrown back to the client. The HelloWorldException extends the normal Java Exception:

HelloWorldException.java

```
package com.spherion.ap.esb.helloworldservice;

import com.spherion.ap.esb.client.EsbException;

public class HelloWorldException extends Exception
{
	public HelloWorldException ()
	{
		super();
	}

	public HelloWorldException (String arg0)
	{
		super(arg0);
	}

	public HelloWorldException (Throwable arg0)
	{
		super(arg0.getMessage(),arg0);
	}

	public HelloWorldException (String arg0, Throwable arg1)
	{
		super(arg0, arg1);
	}
```

```
}
```

3.2 Configuring FreeSB To Recognise New Services

To configure FreeSB to accept a service as being valid, there are two things required:

- Configure a new "user" to represent the service, i.e. create an entry for the service in LDAP.
- Add this entry to the LDAP role designated for all "services".

To do this, the "SupervisorWeb" application is used. Please go to the following URL, noting that the host is where the FreeSB SupervisorService has been deployed to, and the port is what the container has been configured to use:

http://<host>:<port>/SupervisorService/

The following URL would be typical:

http://localhost:8080/SupervisorService/

When FreeSB was installed, the "Administrator" account that was specified in the build.properties file should be familiar to the reader. Please log into the SupervisorWeb using the username and password specified for this Administrator account. The default username is usually "admin".

Once logged in, please proceed to the "Modify Config" section of the application. A list of all currently configured services should be displayed.

In the section down below, where it says "To add a service…", please enter the following details to create the HelloWorld service - note that usually most organizations use a different structure for the LDAP Distinguished Name, i.e. a different organizational unit ("ou") value etc, please modify as necessary, it is only the service name, username and password that is important:

Name of Service: **"HelloWorld"**
Distinguished Name: **"cn=Hello World,ou=Technology,o=Spherion"**
Username: **"helloworldusername"**
Password: **"helloworldpassword"**

A new "HelloWorld" entry should appear in the list of configured services. Note that if the HelloWorld service's link is clicked on, configuration entries should be seen for it's username and password – more configuration entries can be added here but that will be dealt with later.

Next, click on the "System" link in the list. This contains the global properties for FreeSB. Two entries are required, "helloworldpath" and "helloworldport". These two entries describe the path on the web server to get to the service, as well as the

port that the server is running on. Down the bottom, where it says “To create a new config property…” please enter the following two properties (note that the implication of these two properties will be explained in further sections) and hit “Create config”:

Key: **helloworldpath**
Value: **TestService/services/HelloWorld**

Key: **helloworldport**
Value: **8080**

What all this has effectively accomplished is that the service has been configured in FreeSB. When an EsbClient is instantiated by passing in the name of the service as a parameter (i.e. “HelloWorld”), it will recognise the service name and proceed to authenticate that client against FreeSB (using the username and password configured above).

3.3 Making Axis recognise the Adaptor as a SOAP Service

Now that the service implementation, adaptor and custom exception classes have been written, Apache Axis needs to be configured so that this new service can be accessed by SOAP - i.e. turn it into a “Web Service”. It’s the adaptor class that must be exposed as the entry point – it receives the incoming requests first, doing authentication etc *before* passing the request onto the service implementation.

As the Axis documentation indicates, in order to expose a class using SOAP, a WSDD (Web Service Deployment Descriptor) file must be written for it, called “server-config.wsdd”. For this simple adaptor, the actual service declaration is fairly trivial (although it does get more complicated when serialization of custom objects over SOAP is required). An example is listed below:

server-config.wsdd

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment name="defaultClientConfig"
        xmlns="http://xml.apache.org/axis/wsdd/"
        xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
        xmlns:handler="http://xml.apache.org/axis/wsdd/providers/handler">

 <globalConfiguration>
  <requestFlow>
   <handler type="java:org.apache.axis.handlers.JWSHandler">
     <parameter name="scope" value="session"/>
   </handler>
   <handler type="java:org.apache.axis.handlers.JWSHandler">
     <parameter name="scope" value="request"/>
     <parameter name="extension" value=".jwr"/>
   </handler>
  </requestFlow>
</globalConfiguration>
```

```
<handler type="java:org.apache.axis.handlers.http.URLMapper" name="URLMapper"/>
<handler type="java:org.apache.axis.transport.local.LocalResponder" name="LocalResponder"/>
<handler type="java:org.apache.axis.handlers.SimpleAuthenticationHandler" name="Authenticate"/>

<service name="AdminService" provider="java:MSG">
 <namespace>http://xml.apache.org/axis/wsdd/</namespace>
 <parameter name="allowedMethods" value="AdminService"/>
 <parameter name="enableRemoteAdmin" value="false"/>
```

```
 <parameter name="className" value="org.apache.axis.utils.Admin"/>
</service>

<service name="Version" provider="java:RPC">
 <parameter name="allowedMethods" value="getVersion"/>
 <parameter name="className" value="org.apache.axis.Version"/>
</service>

<service name="HelloWorld" provider="java:RPC" style="rpc" use="encoded">

<parameter name="className"
value="com.spherion.ap.esb.helloworldservice.HelloWorldAdaptor"/>
     <parameter name="allowedMethods" value="*"/>
     <parameter name="scope" value="Application"/>
  <typeMapping
       xmlns:ns="http://client.esb.ap.spherion.com"
       qname="ns:EsbException"
       type="java:com.spherion.ap.esb.helloworldservice.HelloWorldException"
       serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
       deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     />
     <typeMapping
       xmlns:ns="http://client.esb.ap.spherion.com"
       qname="ns:EsbPrincipal"
       type="java:com.spherion.ap.esb.client.EsbPrincipal"
       serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
       deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     />
 </service>

<transport name="http">
 <requestFlow>
  <handler type="URLMapper"/>
  <handler type="java:org.apache.axis.handlers.http.HTTPAuthHandler"/>
</requestFlow>
</transport>

<transport name="local">
<responseFlow>
  <handler type="LocalResponder"/>
```

```
 </responseFlow>
</transport>

</deployment>
```

Most of the WSDD file contains standard Axis settings that can be left alone, except for where it's in **bold**, where the service is actually declared. It says that "HelloWorld" is the name of the SOAP service (this effects the path on the web server used to access this service, i.e. Axis will name the last part of the service URL according to this value), that the class exposed should be "HelloWorldAdaptor", that all methods within the class should be made available and that the scope is "Application" - meaning only one instance of the adaptor will be instantiated, and that object will be re-used for every call that comes in. There's also what's known as "typeMapping" which tell Axis that these particular objects can be serialized into SOAP, again refer to the Axis documentation on typeMappings.

Next, the web.xml file is required which tells Jetty that this is a web application:

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app    PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.2//EN"
   "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>

<display-name>Apache-Axis</display-name>
 <servlet>
  <servlet-name>AxisServlet</servlet-name>
  <display-name>Apache-Axis Servlet</display-name>
  <servlet-class>
     org.apache.axis.transport.http.AxisServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
 </servlet>

 <servlet>
  <servlet-name>AdminServlet</servlet-name>
  <display-name>Axis Admin Servlet</display-name>
  <servlet-class>
     org.apache.axis.transport.http.AdminServlet
  </servlet-class>
  <load-on-startup>2</load-on-startup>
 </servlet>

 <servlet-mapping>
```

```
  <servlet-name>AxisServlet</servlet-name>
  <url-pattern>/servlet/AxisServlet</url-pattern>
 </servlet-mapping>

 <servlet-mapping>
  <servlet-name>AxisServlet</servlet-name>
  <url-pattern>*.jws</url-pattern>
 </servlet-mapping>

 <servlet-mapping>
  <servlet-name>AxisServlet</servlet-name>
  <url-pattern>/services/*</url-pattern>
 </servlet-mapping>



 <session-config>
      <session-timeout>30</session-timeout>
  </session-config>

<mime-mapping>
  <extension>wsdl</extension>
   <mime-type>text/xml</mime-type>
 </mime-mapping>

 <mime-mapping>
  <extension>xsd</extension>
  <mime-type>text/xml</mime-type>
 </mime-mapping>

   <welcome-file-list>
      <welcome-file>index.jsp</welcome-file>
      <welcome-file>index.html</welcome-file>
      <welcome-file>index.htm</welcome-file>
   </welcome-file-list>
</web-app>
```

What this does is basically load the Axis Servlet that acts as the engine for all SOAP requests that come in and forwards onto the appropriate adaptor depending on what service was requested.

3.3 Configuring The Service To Initialize On Startup

The service needs to "initialise" on startup, that is, register with FreeSB. This is done by having an "InitService" servlet class that's loaded on start-up by the Jetty container, and instantiates the "HelloWorldAdaptor". This causes the constructor for "HelloWorldAdaptor" in turn instantiating an instance of the EsbClient with the name "HelloWorld" passed into it – hence registering with the FreeSB. Due to timing issues, the "InitService" class should wait until Jetty has completed startup and is accepting incoming connections, so a thread needs to be spawned that will attempt to connect to it's localhost until finally a connection gets through, and only then instantiate "HelloWorldAdaptor".

InitService.java

```
package com.spherion.ap.esb.helloworldservice;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.Socket;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;

public class InitService extends HttpServlet implements Runnable
{
	public void init() throws ServletException
	{
		new Thread(this).start();
	}

	public void run()
	{
		connectToHost();
	}

	public void connectToHost()
	{
		try
		{
			Socket s = new Socket();
			InetSocketAddress isa = new InetSocketAddress("localhost", 8080);
			s.connect(isa);
			s.close();
			new HelloWorldAdaptor();
		}
		catch (IOException ioe)
		{
			System.out.println("Container on localhost not responding, will try
again");
			try
			{
				Thread.sleep(5000);
			}
			catch (InterruptedException ie)
			{
				ie.printStackTrace();
			}
			connectToHost();
		}
	}

}
```

The web.xml also needs to be configured to load this “InitService” servlet on startup, the following lines after the last servlet declaration will work:

web.xml (portion)

```
<servlet>
 <servlet-name>InitService</servlet-name>
 <display-name>Init service</display-name>
 <servlet-class>
  com.spherion.ap.esb.helloworldservice.InitService
```

```
  </servlet-class>
  <load-on-startup>2</load-on-startup>
</servlet>
```

3.4 Deploying The New FreeSB Service

That's all that's required for a FreeSB service. It's time to package the service into a deployable WAR (Web Application Resource) file. Here are some pointers when assembling the WAR file:

- Name the WAR file as "TestService.war" as by default Jetty gets the root directory for a web application by the name of the WAR file, so "TestService.war" will mean the root is "TestService" – this is what FreeSB was previously configured to use as the path to access the service e.g. "helloworldpath".
- Use Apache Ant's "WAR" task to build the WAR file – it's much easier.
- Make sure the "server-config.wsdd" file is placed in the WEB-INF directory.
- Make sure all the files in the "lib" directory of the EsbCommons project is copied to the "WEB-INF/lib" directory.

FreeSB has already written all the classes and deployment descriptors required for this example, please see the "TestService" directory in the distribution for the source code (see the package "com.spherion.ap.esb.helloworldservice") and configuration files. The build script (which can also be used as a template for automating the deployment of custom services) included in the TestService's base directory will deploy the TestService with the following command:

ant all

Note that the SupervisorService, AuthService and EsbLogger must have already been deployed either on the same container or on different containers on different machines – refer to the installation guide for details on deploying FreeSB correctly.

Make sure that the properties for the HelloWorld service have been configured as described in section 3.2.

Please ignore the code in the "com.spherion.ap.esb.client" and "com.spherion.ap.esb.testservice" packages for now – it will be used to describe client functionality in later sections.

Once the application has been deployed, start Jetty (if using the FreeSB build script, can be done using the ant task "start_jetty" – refer to the installation guide) and navigate to the following URL (where <host> is the location of Jetty, <port> is the port it is running on and <root> is what the WAR file was named):

http://<host>:<port>/<root>/services/HelloWorld

A typical URL would look like this:

http://localhost:8080/TestService/services/HelloWorld

This should show "And now... Some Services" with a list of Web Services available, including "HelloWorld". Clicking on the "wsdl" link next to the service name will show the WSDL (Web Service Description Language) for the service. Please see the next chapter for more information on WSDL.

4.0 Writing The Client

Now that the service has been deployed, a client must be written that can access this service. First, the "stubs" must be created, which are a bunch of classes generated by Axis that hide the creation of SOAP messages from the developer, instead exposing standard Java calls. These stubs are generated from the WSDL referred to in the previous section. Then a client needs to be written (as a class that extends the EsbClient) which can use the stubs. Note that if the intent is to store an instance of this client in a HTTP session or persisting it somehow, then make sure all fields within it are serializable.

4.1 Creating The Stubs

The easiest way to create stubs is to use Axis's "WSDL2Java" task. What this does is parse the WSDL for a service, and creates stub classes that have method calls which match that of the service, in this case, the getMessage(String name) method. When these methods are invoked on the stubs, they serialize the calls into SOAP and make the request to the service.

For the HelloWorld service, the basic invocation form for the "WSDL2Java" task looks something like the following (making sure to change <host>, <port> and <root> as appropriate, as well as adding to the classpath from EsbCommons\lib the following jars: axis.jar, saaj.jar wsdl4j.jar, jaxrpc.jar,commons-logging.jar,commons-discovery.jar).

For example:

java -cp axis.jar;saaj.jar;wsdl4j.jar;jaxrpc.jar;commons-logging.jar;commons-discovery.jar org.apache.axis.wsdl.WSDL2Java -p com.spherion.ap.esb.client http://localhost:8080/TestService/services/HelloWorld?wsdl

The "-p com.spherion.ap.esb.client" tells WSDL2Java to create the stubs in the com.spherion.ap.esb.client package – this is the package that also contains the EsbClient class. All stubs and custom clients (that extend EsbClient) **must** be in that package in order for the various argument types to be compatible between the EsbClient itself and its sub-classed custom clients.

4.2 Using The Stubs

Now that the stubs required for communication with the service have been created, it's time to write the client that will use these stubs. It should be called

"HelloWorldClient". This client must be in the same package as the stubs generated previously to ensure compatibility between various classes, so the package "com.spherion.ap.esb.helloworldclient" is what should be used. The client also needs to use FreeSB functionality such as authenticating users etc, therefore the generic EsbClient must be extended. Make sure to code the custom client so that on instantiation, as an argument to the constructor it will accept the service it is acting on behalf of, i.e. if "Workflow Application" wishes to use the HelloWorld service, then the Workflow Application will instantiate an instance of the HelloWorldClient by passing in (say) "WorkflowApp" to the constructor. Of course, "WorkflowApp" needs to be defined as a service within the FreeSB. When the HelloWorldClient gets the name of the service it is acting on behalf of, it should instantiate it's EsbClient super-class with that name as well.

Within the client, the EsbClient superclass is asked to return a URL of where the HelloWorld service is located, passing in the name of the service, and the property keys for the path and port (refer back to section 3.2 where the two properties were created, the keys being "helloworldpath" and "helloworldport").
The getServiceURL(…) method is used to return the desired URL, it will cause the EsbClient to connect to the SupervisorService and ask it for a list of service instances (if any), and will return to the custom client a single URL for a service instance. The URL will be dependant on current load in the system – the EsbClient will perform automatic load balancing. Hence repeated calls may return different URLs that refer to difference service instances. This URL is used to create an instance of the stub and store it in a class variable. To create the stub, a "HelloWorldServiceLocator" object needs to be instantiated which when the getHelloWorld(…) method is invoked, passing in the URL, it will return a stub that will talk to that relevant service instance.

The only thing left to do now is to write a method in the client that corresponds to the getMessage(String name) method in the service. This method will first get a "sequence" number from the super-class. This sequence number is a unique identifier for each message across FreeSB, it acts as a way for messages to be tracked and, in the cases of asynchronous communication, to identify the replies. Next, the method will need to get the user principal and service principal from the super-class. The idea here is, the application that uses this custom client will authenticate against FreeSB when it instantiates it (that "service principal" will be stored in the super-class) and the application should then authenticate the user by invoking the authentication method in the super-class (see Appendix A for a list of EsbClient methods).

HelloWorldClient.java

```
package com.spherion.ap.esb.client;

import com.spherion.ap.esb.client.EsbClient;

public class HelloWorldClient extends EsbClient
{
	private HelloWorldSoapBindingStub hwsbs;
	private String hwurl;

	public HelloWorldClient(String serviceName) throws Exception
	{
```

```
                    super(serviceName);
                    hwurl = super.getServiceURL("HelloWorld", "helloworldpath","helloworldport");
              hwsbs = (HelloWorldSoapBindingStub) new HelloWorldServiceLocator()
                                                    .getHelloWorld( new URL(hwurl));
        }

        public String getMessage(String name) throws Exception
    {
                String seqNo = super.getNextSeqNo();
            EsbPrincipal[] principals = super.getPrincipals();
            hwsbs.getMessage(seqNo, principals, name);
    }
}
```

4.3 Adding Fail-Over To The Client

Refer back to the HelloWorldException that is thrown by the service when the requesting user and service isn't authenticated. This is an application-level exception, meaning that the service is running correctly - the exception was deliberately caused by the business process that mandate all users of the HelloWorld service must be authenticated.

There is also what's known as a "system" exception, where the service fails because of an event outside of the scope of the application, i.e. the container goes down and the host is unreachable. It would be good for a client to trap these system exceptions when they occur and then tell FreeSB to de-register that service and then attempt to connect to another instance of that service (if one exists). This prevents other clients wasting time in trying to connect to the dead service.

All that is required is to code the client in such a way that if an exception occurs when using the service, it is checked to determine whether it's an application level exception or a system level exception. If the former, merely pass the exception on to the application using the client to be dealt with, if it's the latter, then tell the FreeSB to de-register the service that has failed, then get a new URL for another instance of that service (if one exists) from the EsbClient and reconnect to that one. This requires a new method called "connect()" which is called from the catch clause.

Unfortunately Apache Axis is extremely buggy when it comes to exceptions. That is, even though the service might be throwing "HelloWorldException", it arrives at the client as a "RemoteException" – no matter what. Even if the service is down, then the exception that comes back is a "RemoteException". So, the only way to work out what exception is being thrown by the service, is by doing a string comparison on the message field within the RemoteException. This is done using the getMessage() method on the RemoteException, and check to see if string that is equal to the application-level exception thrown by the service (say, HelloWorldException). If it is, simply pass the exception back to the application that's using the client, otherwise, deregister the "dead" service (by calling on an EsbClient method called "deregister()", and try and connect to another one.

HelloWorldClient.java

```
package com.spherion.ap.esb.client;

import com.spherion.ap.esb.client.EsbClient;

import java.net.URL;
import java.rmi.RemoteException;

public class HelloWorldClient extends EsbClient
{

   private HelloWorldSoapBindingStub hwsbs;
   private String hwurl;

   public HelloWorldClient(String serviceName) throws Exception
   {
      super(serviceName);
      connect();
   }

   public void connect() throws Exception
   {
      hwurl = super.getServiceURL("HelloWorld", "helloworldpath",
            "helloworldport");
      hwsbs = (HelloWorldSoapBindingStub) new HelloWorldAdaptorServiceLocator
()
            .getHelloWorld(new URL(hwurl));
   }

   public void deregister() throws Exception
   {
      super.deregister("HelloWorld", hwurl);
   }

   public String getMessage(String name) throws Exception
   {
      try
      {
         String seqNo = super.getNextSeqNo();
         EsbPrincipal[] principals = super.getPrincipals();
         hwsbs.getMessage(seqNo, principals, name);
      }
      catch (HelloWorldException hwe)
      {
         throw new Exception(hwe);
      }
      catch (RemoteException re)
      {
         String errMsg = re.getMessage();
         if (errMsg.indexOf("HelloWorldException") != -1)
         {
            throw new Exception(re);
         }
               re.printStackTrace();
         deregister();
         connect();
         getMessage(name);
```

```
    }
  }
}
```

4.4 Running The Example

The "HelloWorldClient" class has already been written and incorporated into the TestService, under the package "com.spherion.ap.esb.client". Please note that the rest of the classes there are the stubs referred to in section 4.1, they can be over-written if the reader wishes to generate the stubs again.

A simple servlet has also been written that will invoke the "HelloWorldClient" class through a web interface. This servlet is in a class named "EsbClientTestServlet" located in the com.spherion.ap.esb.testservice package. Quite simply, all the servlet does is use the "HelloWorldClient" to authenticate a user, and then invokes the getMessage(…) method.

Please navigate to the following URL to access the web interface, making sure to change <host>, <port> and <root> as appropriate:

http://<host>:<port>/<root>/HelloWorldTestServlet

Typically, the URL would look like this:

http://localhost:8080/TestService/HelloWorldTestServlet

At the form presented, log in using any user account – typically the Administrator account referred to in section 3.2, but other user accounts can be created by using the Supervisor GUI (see section 6.0):

Once in, type in the reader's first name into the field and press the "Get Message" button. This will trigger the HelloWorldClient to call the HelloWorldService and display the message on the screen.

5. Aysnchronous Call-backs

Asynchronous call-backs are relatively straight-forward to perform using FreeSB, with a few caveats. Asynchronous calls result through co-operation between two services, generally no anonymous calls from a service can be made on another service that exposes an asynchronous method – the service being called must know which specific service it should in turn send the response to. There is no way (at present) to dynamically state in the request which service the response goes to.

Consider two services, service A and service B. If service A wishes to asynchronously call service B, then service B needs to have an asynchronous method written in such a way that after processing the request, service B will know to send the response back to service A specifically. Likewise, service A needs to have a corresponding call-back method that service B can invoke once service B has finished processing. For service B to invoke this call-back method, service A's client stubs must be available to service B, and of course service B's client stubs must be available to service A to make the original call in the first place.

<<insert diagram>>

Recall that a service requires two classes, the service implementation and adaptor. With regard to asynchronous calls, it is up to the adaptor to co-ordinate such activity (after all, it is the adaptor that is exposed to SOAP, not the implementation class). When an asynchronous call arrives at the adaptor, the adaptor should (after checking the security credentials of course) add that request to some sort of queue (say a JMS queue) or even spawn off a new thread to do the work – whichever way, the method should return immediately after so the client isn't blocked. That's the whole point of asynchronous calls.

For clarity, asynchronous methods in a service should be prefixed with "asynch". For example:

public void asynchRegisterService(…)

or for getter/setter methods:

public void getAsynchMessage(…)

Likewise, for call-back methods in the client, they should be prefixed with "callback". For example:

public void callbackRegisterService(…)

or for callback getter/setter methods:

public void callbackGetAsynchMessage(…)

5.1 Implementing Asynchronous Methods

In order to show how asynchronous call-backs work involve tweaking the existing code slightly. Because this is just an example, there is no need to go to the effort of writing a brand new service to represent the calling service, instead the existing HelloWorld service can be used to asynchronously call itself – that is, the sample test servlet will call an asynchronous method on the HelloWorld service, which

will then use it's own HelloWorldClient to call another method within itself (the "callback" method). This callback method will simply update some data in the test servlet (since they are all in the same application). This will let the reader become familiar enough with asynchronous communication to proceed further. To explain the scenario, the following diagram shows what will happen:

<<insert diagram of helloworldservice calling itself here>>

A new method is added to the HelloWorld service's adaptor (HelloWorldAdaptor.java), called "getAsynchMessage(…)" with the arguments identical to the existing "getMessage(…)" method and it will even invoke the same method in the service implementation class ("HelloWorld.java"). The difference is that this new asynchronous method's return type will be void (since it does not return any data within the same call, only later).

HelloWorldAdaptor.java (excerpt)

```
    public void getAsynchMessage(String esbSeqId, EsbPrincipal[] principals,
          String name) throws Exception,HelloWorldException
          {
       //have to convert the arguments to finals in order to be used in inner-classes
       final String esbSeqIdUsedInsideThread = esbSeqId;
       final String nameUsedInsideThread = name;

       //checks the security credentials of the requesting user and service
          if(ec.checkPrincipals(principals))
          {
             //starts a thread from the anonymous inner class to perform the
             //asynchronous operation
             new Thread()
             {
                public void run()
                {
                   String msg = hw.getMessage(nameUsedInsideThread);
                   try
                   {
                      HelloWorldClient threadHwc = new HelloWorldClient("HelloWorld");
                      EsbPrincipal[] servicePrincipal = threadHwc.getPrincipals();
                      //wait for 20 seconds before calling back
                      sleep(20000);
                      threadHwc.callbackGetAsynchMessage
(esbSeqIdUsedInsideThread,servicePrincipal,msg);
                   }
                   catch (Exception e)
                   {
                      e.printStackTrace();
                   }

                }
             }.start();

             return;

          }
          else
          //if it was invalid, throw an exception back to the client
          {
```

```
            throw new HelloWorldException("One or more principals " +
            "is not valid!");
        }

}
```

As can be seen, the method starts a new thread (the logic of which is defined in an inner-class) and returns immediately. The spawned thread is what tiggers the service implementation, gets the result and after waiting for 20 seconds, uses the HelloWorldClient to invoke the callback method (the callback method will be written in the same class next). Now, with regard to the principals – as this is the service itself making the call, and there is no user who is authenticated, the only principal returned with the getPrincipals() method will be the service principal. The ESB sequence ID (esbSeqId) is of paramount importance with asynchronous calls – that is the only way to track what response is for which request. Make sure the esbSeqId that came along with the request is passed along when making the callback.

5.2 Writing The Callback Method

The callback method is presented below. Recall that for this example, in order to maintain simplicity, the callback method will be in the same class as the asynchronous method.

HelloWorldAdaptor.java (excerpt)

```
        //calls a static method in the HelloWorldTestServlet class which simply
        public void callbackGetAsynchMessage(String esbSeqId, EsbPrincipal[] principals,
                                String msg) throws Exception,HelloWorldException
        {
                HelloWorldTestServlet.responseReceived(esbSeqId, msg);

        }
```

When a callback arrives, the esbSeqId needs to be correlated against some logic that keeps track of the outgoing requests – perhaps there will be a HashSet that stores the esbSeqIds associated with each outgoing request, and when a callback arrives, simply check the esbSeqId that arrived with what is inside the HashSet. In the above code it is doing exactly that, the callback is calling a static method in the HelloWorldTestServlet, passing along the esbSeqId and the actual message. This static method in the servlet simply checks to see if the esbSeqId exists in a HashSet, and if it does, sets the received message as what is to be displayed on the web page.

It would be even better to persist the esbSeqId (and any other request related information) so that request/call-back correlation is not lost when the container goes down. It will also prevent the scenario where a different instance of the same service where to receive a particular callback, but that instance did not originate the request, hence it rejects the callback since it has no record of the esbSeqId in memory. Since when using a persistent database, there would only be one central repository of esbSeqIds and not in memory inside each service, all the instances could receive the callback without a problem – they would simply check the

database for the esbSeqId. Of course, this all relies in making sure all service instances point to the same database.

5.3 Coding The Client For Asynchronous Calls

The two methods, getAsynchMessage(…) and callbackGetAsynchMessage(…) need to be added to the HelloWorldClient. To do this, the updated HelloWorld service needs to be deployed, and the stubs re-generated from it's WSDL (see section 4.0) to reflect the new methods in the service. Once this has been performed, the two methods can be added to the HelloWorldClient that will use the new methods in the stubs:

HelloWorldClient.java (excerpt)

```
    public String getAsynchMessage(String name) throws Exception
    {
       String seqNo = null;
       try
       {
          seqNo = super.getNextSeqNo();
          com.spherion.ap.esb.client.EsbPrincipal[] principals = super.getPrincipals
();
          hwsbs.getAsynchMessage(seqNo, principals, name);
       }
       catch (HelloWorldException hwe)
       {
          throw new Exception(hwe);
       }
       catch (RemoteException re)
       {
          String errMsg = re.getMessage();
          if (errMsg.indexOf("HelloWorldException") != -1)
          {
             throw new Exception(re);
          }
          re.printStackTrace();
          deregister();
          connect();
          getAsynchMessage(name);
       }
       return seqNo;
    }

    public void callbackGetAsynchMessage(String esbSeqId, EsbPrincipal[]
principals,
          String msg) throws Exception
    {

       try
```

```
        {
            hwsbs.callbackGetAsynchMessage(esbSeqId, principals,msg);
        }
        catch (HelloWorldException hwe)
        {
            throw new Exception(hwe);
        }
        catch (RemoteException re)
        {
            String errMsg = re.getMessage();
            if (errMsg.indexOf("HelloWorldException") != -1)
            {
                throw new Exception(re);
            }
            re.printStackTrace();
            deregister();
            connect();
            callbackGetAsynchMessage(esbSeqId, principals, msg);
        }
    }
```

Relatively straightforward, the getAsynchMessage(…) call is almost identical to the way the synchronous getMessage(…) call is made, with the exception that the method returns the esbSeqId, and the actual call on the stub returns no argument (since it is void).

The client code for the callbackGetAsynchMessage(…) is significant in this particular case because (as noted previously) while this particular method has been implemented with fail-over, unless the esbSeqId is stored in a central database somewhere, then if one instance fails and the client fails-over to another instance, then of course any esbSeqId references kept in memory inside the client will be lost. Always persist the esbSeqId (and other such request related data) to a database so that multiple instances of the service can effectively receive callbacks.

5.4 Running The Example

The “EsbClientTestServicesimple servlet referred to in Section 4.4 has been written to invoke the asynchronous method in the “HelloWorldClient” class through the web interface. Look for a class named “EsbClientTestServlet” located in the com.spherion.ap.esb.testservice package. The servlet uses the “HelloWorldClient” to authenticate a user, and then invokes the getAsynchMessage(…) method. When the callback is made (remember, to the same service) the callback method then sets the message to be displayed on the web interface.

Please navigate to the following URL to access the web interface, making sure to change <host>, <port> and <root> as appropriate:

http://<host>:<port>/<root>/HelloWorldTestServlet

Typically, the URL would look like this:

http://localhost:8080/TestService/HelloWorldTestServlet

At the form presented, log in using any user account – typically the Administrator account referred to in section 3.2, but other user accounts can be created by using the Supervisor GUI (see section 6.0):

Once in, type in the reader's first name into the field and press the "Get Asynchronous Message" button. This will trigger the HelloWorldClient to call the HelloWorldService and return. After 20 seconds, the HelloWorldService in turn will use the HelloWorldClient to call the callback method, which comes back to the HelloWorldService. This callback method in the HelloWorldService (as described in section 5.2) will set the display message in the EsbClientTestServlet (since all these classes are in the same application, the callback method can "cheat" and just call the HelloWorldService using a static method).

The "Refresh Page" button can be used to see the displayed message, it will be prefixed with "Asynch: " for clarity.

6.0 The Supervisor GUI

As mentioned in previous sections, the Supervisor GUI can be used to configure and manage FreeSB. It allows the creation of new users, roles and services, as well as editing various properties.

Please navigate to the following URL, noting that the host is where the FreeSB SupervisorService has bee deployed to, and the port is what the container has been configured to use:

http://<host>:<port>/SupervisorService/

When FreeSB was installed, the "Administrator" account that was specified in the build.properties file used to install with should be familiar. Please log into the SupervisorWeb using the username and password specified for this Administrator account. The default username is usually "admin".

Note that any configuration changes will require a restart of all instances of the particular service affected, and if it is a system property that is changed, then **all** services and their instances must be restarted.

Upon logging in, there should be four clickable options visible:

- View Registered Services
- Modify Roles
- Modify Users
- Modify Configuration

6.1 View Registered Services

This page allows the viewing of services currently registered with FreeSB. The name of the service, it's IP address and the time it registered with FreeSB is displayed. It does not matter whether service instances that have been shut-down are still displayed as registered – if a client tries to use the dead service, it will be automatically de-registered as part of FreeSB's fail-over mechanisms.

6.2 Modify Roles

This page allows configuration of the roles various users are assigned to. This includes creating and deleting such roles, as well as assigning various users to each role. It is also possible to view the various permissions associated with a role, i.e. what actions they are allowed to perform within FreeSB. To understand more about permissions, refer to the document titled "FreeSB Permissions".

6.2.1 Creating A New Role

Firstly, to create a new role, the full LDAP "Distuingished Name" (DN) of the role must be entered, i.e it must be fully qualified such as:

cn=User Modifier,ou=Technology,o=Spherion

The reason the full Distuingished Name is required and not just the name of the role ("User Modifier") is that FreeSB does not know the LDAP heirachy of the reader's organization, i.e. there may be a completely different structure than the name, organizational unit and organization as used in the above sample DN.

Once the role's DN is entered into the role name field, then in the next field there is a requirement to assign a new "rank" number to the role. The concept of the rank is explained in the FreeSB Permissions document.

6.2.2 Displaying a User's Roles

It is possible to view all the roles that a particular user belongs to. The information required is the username (not the DN) of that user, and a page will list all the roles that user belongs to. In order to view roles for all users, simply clear the field and submit.

6.2.3 Displaying All Users For a Role

In the list of roles displayed, next to each entry there is a "View Users" link. Clicking on it will display the users that are assigned to that particular role.

6.2.4 Adding A User To A Role

From the page displayed in section 6.2.3, simply enter the username (not the DN) into the field and submit. Note that the user must have already been created; refer to section 6.3 for further information.

6.2.5 Deleting A User From A Role

To delete a user from a role, on the page referred to in section 6.2.3, hit the delete link next to the appropriate user entry.

6.2.6 Viewing Permissions For A Role

To view the access permissions for a Role, on the main "View Roles" page, click on the "View Permissions" link next to the role. A page will be displayed that shows all the permissions that role has, with the mask on the Moan object separated by a colon. For example:

a:/SupervisorService/modifyUsers

The mask here is "a" and the Moan object is "modifyUsers" which sits under its parent Moan object "SupervisorService". Refer to the FreeSB Permissions document on what masks and Moans are.

6.2.7 Adding a Permission To a Role

On the page referred to in section 6.2.6, the mask can be entered into the relevant field at the bottom of the page. This mask can be singular or multiple – i.e. "t" will assign the permission "t" to that Moan object, whereas "ty" will assign the permissions "t" and "y" to that Moan object.

The Moan object name must also be entered into the relevant field, noting that all Moan objects begin with a forward slash "/" and a Moan Object, with optionally any number of child Moan objects, i.e. "/SupervisorService" is fine, along with "/SupervisorService/modifyUsers", as well as "/SupervisorService/modifyUsers/modifyUsernameAttribute" etc.

6.2.8 Deleting A Permission

On the page referred to in section 6.2.6, click on the delete link next to the permission entry.

6.2.8 Deleting A Role

On the main "Mdoify Roles" page, simply click on the "Delete" link next to the appropriate role. This will completely delete that role from FreeSB.

6.3 Modify Users

This page allows the management of FreeSB users, including creating, deleting and searching.

6.3.1 Create a New User

In the form at the bottom of the "Modify Users" page, please enter and submit the details of the user to be created, including the Username, the full LDAP Distuingished Name (DN), the last name and finally the password.

For example:

Username: **johnsmith**
Distuingished Name: **cn=John Smith,ou=Technology,o=Spherion**
Last Name: **Smith**
Password: **firefox**

The reason the full Distuingished Name is required and not just the name of the user ("John Smith") is that FreeSB does not know the LDAP heirachy of the reader's organization, i.e. there may have a completely different structure than the name, organizational unit and organization as used in the above sample DN.

6.3.2 Deleting a User

To delete a user, click on the "Delete" link next to the appropriate user entry. This will also automatically remove the user from all the roles they belonged to.

6.3.3 Searching For a User

If there is a significant amount of users in the system the list of users could become very large and trying to find a single user or group of users would be quite difficult. LDAP search queries can be used to narrow the list of users, by using the search form in the middle of the page. For example, to find someone with the common name "John Smith" enter the following search string:

cn=John Smith

or to search for anyone beginning with John:

cn=John*

6.4 Modify Config

This page can be used to modify the various configuration properties used throughout FreeSB. It can also be used to add a new service to FreeSB, which involves creating some basic configuration properties for that service.

6.4.1 Adding a New Service

In the section down below, where it says "To add a service…", please enter the name of the service to create (preferably with no spaces), followed by the LDAP distuingished name and then the username and password. An example is the "HelloWorld" service we added in section 3.2:

Name of Service: **"HelloWorld"**

Distinguished Name: **“cn=Hello World,ou=Technology,o=Spherion”**
Username: **“helloworldusername”**
Password: **“helloworldpassword”**

Next, click on the “System” link in the list. This contains the global properties for FreeSB. Two entries need to be added which define the path on the web server to get to the service, as well as the port that the server is running on. Down the bottom, where it says “To create a new config property…” please enter the properties as required and hit “Create config”. An example is the “HelloWorld” configuration in section 3.2:

Key: **helloworldpath**
Value: **TestService/services/HelloWorld**

Key: **helloworldport**
Value: **8080**

6.4.2 Adding a New Configuration Property

On the main “Modify Config” page click on the “View Config” link next to the name of the service a new property is required for. There is also the “View Config” button next to the “System” entry that defines global, system-level properties.

At the bottom of the page, where it says “To create a new config property…” enter the key and value of the property. Avoid spaces in the key.

6.4.3 Updating / Deleting Configuration Properties

In the page referenced in section 6.4.2, the various properties can be modified by changing the displayed values on the form as appropriate, and hitting the “Update” button. Please be careful when modifying “System” properties as well as properties for the standard FreeSB services (AuthService, SupervisorService, EsbLogger).

To delete a property, click on the “Delete” checkbox next to the particular entries to be deleted, and click on “Update”. Please note that it is not possible to delete certain properties – usually the ones that are associated with the base install of FreeSB, i.e. required for operation. These will have the “Delete” checkbox greyed out.

7.0 Extra FreeSB Features

This section will describe extra functionality that FreeSB offers but have not yet been covered by the preceding sections.

7.1 Accessing Properties From The EsbClient

The EsbClient or a custom extension of it is used to access the various configuration properties in FreeSB, (creating such properties is referred to in section 6.4.2).

A client can only access the properties for the service it was instantiated for. It can however also access “System” properties. When the EsbClient (or an extension of it for talking to custom services) is instantiated, passing in the name of the service it is acting on behalf of to the constructor, the EsbClient automatically retrieves the configuration for that service from the SupervisorService. The EsbClient also retrieves the “System” configuration at the same time. To get a particular property value, the following method would be used on the EsbClient, passing in the key of the property to be retrieved:

public String getProperty(String key)

For example, the following code instantiates the EsbClient, passing in the name of the service it is acting on behalf of (which is “HelloWorld”) to the constructor and finally retrieves the “helloworldpath” property:

```
EsbClient ec = new EsbClient("HelloWorld");
String path = ec.getProperty(“helloworldpath”);
```

7.2 Logging In FreeSB

Events can be logged in FreeSB using the following method on the EsbClient:

public void log(String logLevel,String sourceOrDestination, String message)

The “logLevel” argument is the threshold of the log message; the following values can be used (which are retrievable by static fields in the EsbClient, or an extension of it such as the HelloWorldClient):

HelloWorldClient.LOG_LEVEL_INFO = "INFO"
HelloWorldClient.LOG_LEVEL_DEBUG = "DEBUG"
HelloWorldClient.LOG_LEVEL_ERROR = "ERROR"
HelloWorldClient.LOG_LEVEL_WARN = "WARN"
HelloWorldClient.LOG_LEVEL_FATAL = "FATAL"

The “sourceOrDestination” argument indicates is used to indicate either where an event originated or where an event is going to, i.e. at the client end, it might be used to indicate which service the call is going to (say, “AuthService”), or at the service end, which service the call came from (say, “HelloWorldService”).

The “message” argument is obvious - it’s the text of the log message.

Note that FreeSB itself internally logs certain events.

Appendix A: Commands In The EsbClient

This section describes the base level functionality provided by the EsbClient, and performs all FreeSB related tasks. If an extended clientis used that talks to custom services, these methods can be called using the super keyword argument.

EsbClient.java:

Method	Description
boolean authnUser(String username, String password)	This method authenticates the user, storing the principal in that instance of the EsbClient, returning a true or false if authentication succeeded. There can only be one user authenticated per EsbClient instance, so for every user using an application, there will need to be an EsbClient tracking that user within the session. The EsbClient has been designed so that this will not cause a performance problem – all the system level checks etc are cached.
boolean checkAllowed(String permission, String namespace)	Checks to see if the current user's principal has that particular permission on that namespace (or 'moan') object. Refer to the FreeSB Permissions document for more information on handling permissions.
boolean checkAllowed(EsbPrincipal principal, String permission, String namespace)	Checks to see if the principal passed in principal has that particular permission on that namespace (or 'moan') object. Useful when dealing with multiple principals at the service end, i.e. a method would receive both the service principal and user principal, this checkAllowed method can be used to ensure at least one of those principals has the permission to invoke that method.
boolean checkPrincipals(EsbPrincipals[] principals)	Used to check if multiple principals are valid with a single call, this makes it easier for checking principals that have arrived with a request to a service.

boolean deregister(String serviceName, String serviceURL)	Deregisters a particular service instance from FreeSB. The name of the service and the url it resides at must be provided.
String getNextSeqNo()	Returns a unique sequence number that can be used to track messages across FreeSB. Especially useful for asynchronous communication and also audit trails.
EsbPrincipal[] getPrincipals()	Returns the service principal and user principal, if the user has been authenticated, otherwise returns just the service principal.
String getProperty(String key)	Returns the property value for that particular key. Only the properties for the service that the EsbClient was instantiated with are available.
String getServiceURL(String serviceName,String pathKey,String portKey)	Returns a URL for the service requested in the arguments – the name of the service, the path it sits on the server (usually retrieved via properties) and the port of the container hosting the service (again, retrieved using properties). The EsbClient will contact the SupervisorService and ask for a list of those services, and then the EsbClient will form a load-balanced URL to return.
String getUsername()	Returns the username of the currently authenticated user. If no user has been authenticated, returns null.
void invalidateUser()	Invalidates the current user's principal associated with the EsbClient. Useful with someone pressing a logout button etc.
boolean isValid()	Checks to see if the currently authenticated user's principal is valid.
void log(String logLevel,String sourceOrDestination, String message)	Logs a message to the EsbLogger service, which stores it in a database. Required is the threshold level (INFO, DEBUG etc) and the name of the service being called or name of the service the call came from, as well as the message itself.

Todo – make sure when creating a service that it is added to the service role
Todo – make a separate method called “registerService” so that it will explicitly
Todo – add bit on registering with FreeSB on startup to the writing a service section.
Todo – Add a dynamic client generator.
Todo – what happens with fail over and asynchronous callbacks if the original host is down?
Todo – make creating a service add the part and port properties automatically
Todo – explain that you don’t need to use the custom client at the service end, can just call registerService() method
Todo – appendix with all accessible methods in client including modifying users etc
Todo – move logging from client to service, better performance
Todo – add section to checking permissions
Todo – maybe change it so that the supervisor service will form the url and return it? Maybe bad because of load balancing issues.
Todo – viewing log messages via supervisor gui
Todo – improve performance, cleaned up
Todo - including moving to different soap transport
Todo – write and publish roadmap
Todo – snmp service
Todo - .NET client
Todo – Jboss container

tell the FreeSB to register it as a service.
Gotta include the Axis WSDL descriptor file.
Include bit on watching startup of the container.
Luckily, there’s an automatic tool to do all this.
- Problems with Axis (sending all exceptions as remote exceptions etc)
Securing FreeSB (HTTPS etc)
Secure the Auth Service’s role and user management stuff
- Make sure to configure the build so that the servlet’s deployment URL is the same as each service’s path and port in the config file i.e. tokenise the whole thing
- Make sure to write a method into the EsbClient that will send it’s URL to the service in order to perform the callback.

Load balancing – show them how to swap between services after a call