CHAPTER 28

Web Services and Jini

One of the middleware frameworks being heavily promoted at present is that of Web Services. Web Services are built upon the Simple Object Access Protocol (SOAP) invocation protocol, the Web Services Description Language (WSDL) specification language, and the Universal Description, Discovery, and Integration (UDDI) discovery system. In this chapter, we look how clients and services from different frameworks can interoperate, with particular reference to Web Services and Jini.

Integrating Web Services and Jini

While this book has been about Jini, other middleware systems are in use, such as CORBA, Web Services, UPnP, and Salutation, among many others. While it would be very convenient for software developers if all but their favorite middleware were to disappear, this is unlikely to happen. There are technical and political reasons for many of these frameworks to survive, and so software developers will just have to live in a world of multiple middleware systems.

Users, on the other hand, just want their different pieces of software to work together, no matter what framework is used. It is up to software developers to figure out how to get a Web Service client to work with, for example, a mixture of Jini and UPnP services.

The most common way of getting such mixtures to work is through a *bridge*. That is, to get a Web Service *client* to talk to a Jini *service*, typically a bridge will be an application sitting between them and acting as a Web Service *service* and a Jini *client*. In the middle, the bridge translates from one framework to the other, in both directions, as shown in Figure 28-1.

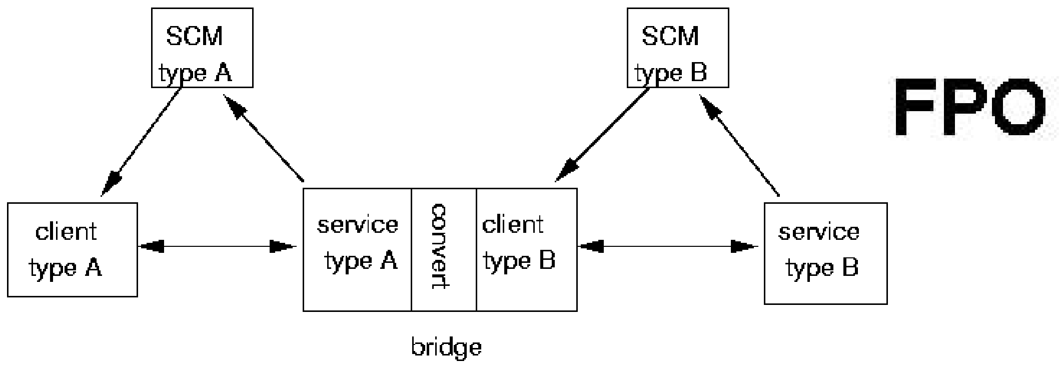

Figure 28-1. *Bridging between middleware systems*

There are two aspects to a bridge: one is concerned with discovery and the other with invocation.

- Discovery allows one middleware client to discover a different middleware service—for example, a CORBA client discovering a Jini service. Typically this is done by the client discovering the bridge and the bridge discovering the service. This may involve two service cache managers (lookup services, name services, etc.).
- The bridge allows one middleware client to make calls on another middleware service—for example, a CORBA client making calls on a Jini service. It will convert calls received under one protocol into calls in the other. This will often involve conversion of data from one format to another.

Web Services and Jini have special features that make this a simpler task than in general:

- Web Services are supposed to use the discovery system UDDI. However, UDDI was designed independently as a global white/yellow/green/blue pages directory, and it turns out to be poorly suited to Web Service discovery. In practice, it seems that most Web Service developers rely on the URLs hard-coded into WSDL files, and don't do any discovery at all.
- In Chapter 1, the section "The End of Protocols" discussed how Jini doesn't really care about invocation protocols, but only about discovery. Of course, a lot of this book has been about how to invoke a service, but much of that discussion is about the choices that a service developer has. The client doesn't care.
- The XML data types don't map directly to Java data types and vice versa. However, there are now standardized mappings with implementations from several vendors.

While building a bridge is in general a nontrivial process, the standardization of data mappings, the indifference of Jini to invocation protocols, and the hard-coded addresses of Web Services simplifies building a Web Services to Jini bridge in the following way:

- Web Services don't need to be discovered; they just need to be looked up. Web clients don't need to do discovery since they have the services' URLs hard-coded in the WSDL document.
- Jini clients and services can talk SOAP (the Web Service protocol) just as easily as they can talk any other invocation protocol. The client doesn't even know what invocation protocol is used, while the service programmer just has to SOAP-enable the services.
- Jini clients and services can make use of existing libraries to handle SOAP queries.

The case of a Jini client talking to a Web Service can lead to several models, as shown in the following figures. In Figure 28-2, the proxy can be an ordinary (e.g., Jeri) proxy talking back to its service. This service also acts as a Web Service client.

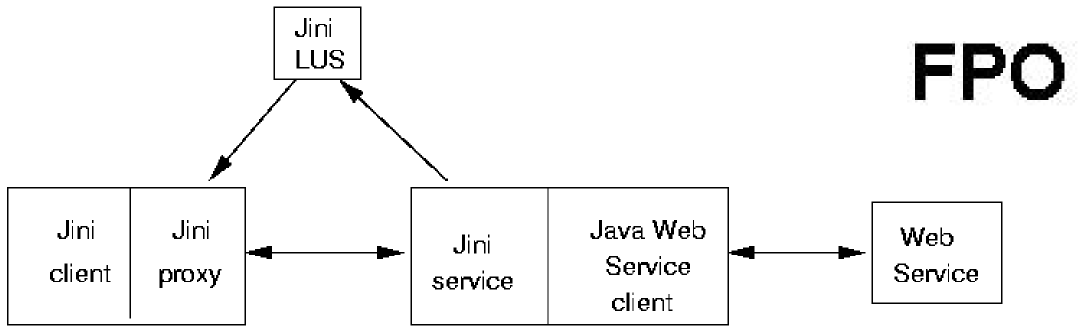

Figure 28-2. *Bridging between a Jini client and a Web Service*

Figure 28-3 shows a “smart” proxy that talks directly to the Web Service.

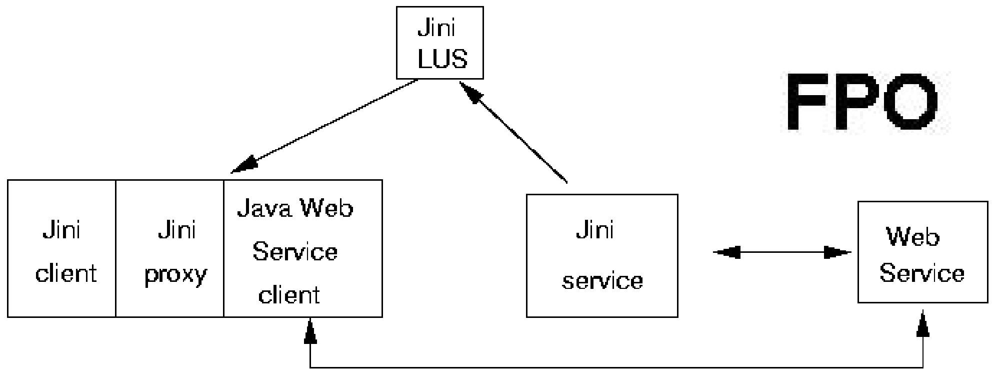

Figure 28-3. *Smart proxy bridging between a Jini client and a Web Service*

Simple Web Service

I’ll illustrate this example with a simple Web Service, for file classification again. To avoid the complexities of Web Service types and with deployment of such services, let’s simplify the service to one that takes a string as file name and returns the MIME type as a string `major/minor`. The class is not in a package, allowing simple deployment under Apache Axis. The implementation of this service is then straightforward:

```
/**
 * FileClassifierService.java
 */
public class FileClassifierService {
    public String getMIMEType(String fileName) {
        if (fileName.endsWith(".gif")) {
            return "image/gif";
        } else if (fileName.endsWith(".jpeg")) {
            return "image/jpeg";
        } else if (fileName.endsWith(".mpg")) {
            return "video/mpeg";
        } else if (fileName.endsWith(".txt")) {
            return "text/plain";
        } else if (fileName.endsWith(".html")) {
            return "text/html";
        } else
            // fill in lots of other types,
            // but eventually give up and
            return "";
    }
    public FileClassifierService() {
        // empty
    }

} // FileClassifierService
```

The Apache Axis server runs under Apache Tomcat and is a popular means of delivering Web Services written in Java. It includes libraries for both the client side and service side. The simplest way of deploying the service under Axis is to copy the implementation source code to the `axis/webapps` directory, renaming the extension `.jws` instead of `.java`.

The service can, of course, be written in many different languages. This is usually done by a horrible practice that has become common with Web Services: reverse engineer the implementation given previously to a WSDL specification, and then forward engineer this to your favorite language. We will ignore all such issues here.

On the client side, a consumer of this service can then be written most simply as follows:

```
package ws;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import javax.xml.namespace.QName;
import ws.MIMEType;
public class TestWSClient {
    public static void main(String [] args) {
        try {
            String endpoint =
                "http://localhost:8080/axis/FileClassifierService.jws";

            Service  service = new Service();
            Call     call    = (Call) service.createCall();

            call.setTargetEndpointAddress( new java.net.URL(endpoint) );
            call.setOperationName(new QName("http://soapinterop.org/", "getMIME-
Type"));

            String ret = (String) call.invoke( new Object[] { "file.txt" } );
            System.out.println("Type of file 'file.txt' is " + ret);
        } catch (Exception e) {
            System.err.println(e.toString());
        }
    }
}
```

There are other ways of achieving the same result, but this is good enough for the rest of this chapter, which is intended to show how Jini and Web Services can interoperate rather than delve into the arcanities of Web Services.

Bridging Between a Jini Client and Web Service, Example 1

A bridge that acts as a Jini service implementing the `common.FileClassifier` specification used throughout this book, and also as a client to the previous file classification Web Service, can be written by essentially including the Web Service client code from earlier into the implementation of the Jini service methods. The bridge is a normal Jini server advertising the following Jini service implementation:

```
package ws;
import common.MIMEType;
import common.FileClassifier;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import javax.xml.namespace.QName;
/**
 * FileClassifierImpl.java
 */
public class FileClassifierImpl implements RemoteFileClassifier {

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException {
        try {
            String endpoint =
                "http://localhost:8080/axis/FileClassifierService.jws";

            Service  service = new Service();
            Call     call    = (Call) service.createCall();

            call.setTargetEndpointAddress( new java.net.URL(endpoint) );
            call.setOperationName(new QName("http://soapinterop.org/", "getMIME-
Type"));

            String ret = (String) call.invoke( new Object[] { fileName } );
            return new MIMEType(ret);
        } catch (Exception e) {
            throw new RemoteException("SOAP failure", e);
        }
    }
    public FileClassifierImpl() throws java.rmi.RemoteException {
        // empty constructor required by RMI
    }

} // FileClassifierImpl
```

This service can export a Jeri or RMI proxy to a Jini client as we have seen before. Client calls on the proxy are sent to this service, which acts as a Web Service client using the model of Figure 28-2. When this implementation is built and run, it will need the Axis libraries on the Jini service side.

Bridging Between a Jini Client and Web Service, Example 2

A service can be written that follows the second pattern in Figure 28-3, simply by changing the inheritance from `RemoteFileClassifier` to `FileClassifier` and `Serializable`. A client then gets a copy of this service and all calls are made locally in the client.

```
package ws;
import common.MIMEType;
import common.FileClassifier;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import javax.xml.namespace.QName;
/**
 * FileClassifierImpl.java
 */
public class FileClassifierSerializableImpl implements FileClassifier,
                                                       java.io.Serializable {

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException {
        try {
            String endpoint =
                "http://localhost:8080/axis/FileClassifierService.jws";

            Service  service = new Service();
            Call     call    = (Call) service.createCall();

            call.setTargetEndpointAddress( new java.net.URL(endpoint) );
            call.setOperationName(new QName("http://soapinterop.org/", "getMIME-
Type"));
```

```
            String ret = (String) call.invoke( new Object[] { fileName } );
            return new MIMEType(ret);
        } catch (Exception e) {
            throw new RemoteException(e);
        }
    }
    public FileClassifierImpl() throws java.rmi.RemoteException {
        // empty constructor required by RMI
    }

} // FileClassifierImpl
```

This has major implications for the classes downloaded to the client! In order to run this service on the client side, it needs access to the class files for the Axis classes `Call`, `Service`, and `QName`. The client cannot be expected to have these, since it doesn't need to know any details of the implementation. So the Axis libraries have to be placed on an HTTP server and listed in the Jini server's codebase. The libraries are over 1MB in size, so needing to send these across the network can result in a substantial download to the Jini client.

Bridging Between a Web Service Client and Jini Service

Forming a bridge between Web Service clients and Jini services follows the same general structure, as shown in Figure 28-4.

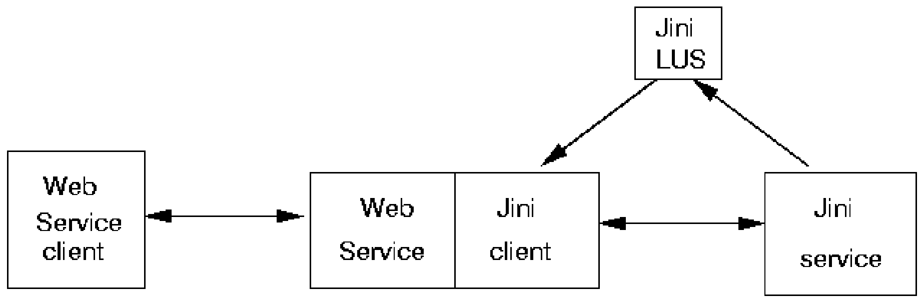

Figure 28-4. *Bridge between a Web Service client and a Jini service*

In this case, we put the Jini client code into the Web Service implementation.

There are a couple of wrinkles in getting this to work properly with Apache Axis and the Tomcat server:

- Jini class files
- Security policy

The Jini class files are not in the classpath for Tomcat. However, Tomcat and Apache allow any extra `.jar` files required by a Web Service to be placed in special `lib` directories under the service's WEB-INF directory. Copying the Jini files jsk-lib.jar and jsk-platform.jar to this directory will make them available to the Web Service. This will be part of the deployment mechanisms for the Web Service.

The issue of a security policy is potentially more difficult. A server such as Tomcat can be started either with or without a security manager. If it uses a security manager, then the default security policy does not allow a new security manager to be put in place. This blocks a Jini client from installing an RMISecurityManager, and so it cannot download a Jini registrar and find Jini services. Negotiation would then be required with the Tomcat administrator to add in sufficient permissions to the security policy to allow a Jini client to run.

If Tomcat is run without a security manager, then it is possible for the Web Service to install one. But it will then need to use a security policy. Up to now we have specified such a policy as a command-line argument, but the command line is not accessible to an Axis Web Service. The workaround is to use System.setProperty() to set the security policy file before installing a security manager.

All I/O to the console has to be cleaned up and put into remote exceptions. With these factors, the Web Service to bridge to a Jini service looks like this:

```java
/**
 * FileClassifierJiniService.java
 */
import common.FileClassifier;
import common.MIMEType;
import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ServiceDiscoveryManager;
import net.jini.core.lookup.ServiceItem;
import net.jini.lease.LeaseRenewalManager;
import java.rmi.RemoteException;
public class FileClassifierJiniService {
    private final static long WAITFOR = 10000;
    public String getMIMEType(String fileName)  throws RemoteException {

        ServiceDiscoveryManager clientMgr = null;
        // set a security policy file here since we don't have command-line access
        System.setProperty("java.security.policy",
                           "/home/httpd/html/java/jini/tutorial/policy.all");
        System.setSecurityManager(new RMISecurityManager());
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null, // unicast locators
                                           null); // DiscoveryListener
            clientMgr = new ServiceDiscoveryManager(mgr,
                                                    new LeaseRenewalManager());
        } catch(Exception e) {
            throw new RemoteException("Lookup failed", e);
        }


        Class [] classes = new Class[] {FileClassifier.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);
        ServiceItem item = null;
        // Try to find the service, blocking until timeout if necessary
        try {
            item = clientMgr.lookup(template,
                                    null, // no filter
                                    WAITFOR); // timeout
        } catch(Exception e) {
            throw new RemoteException("Discovery failed", e);
        }
        if (item == null) {
            // couldn't find a service in time
            return "";
        }
        // Get the service
        FileClassifier classifier = (FileClassifier) item.service;
        if (classifier == null) {
            throw new RemoteException("Classifier null");
        }
        // Now we have a suitable service, use it
        MIMEType type;
        try {
            type = classifier.getMIMEType(fileName);
            return type.toString();
        } catch(java.rmi.RemoteException e) {
            throw e;
        }
    }
    public FileClassifierJiniService() {
        // empty
    }

} // FileClassifierJiniService
```

The steps to get all this running are as follows:

1. Download and install Apache Tomcat and Axis.
2. Edit the `FileClassifierJiniService.java` file to point to a valid security policy file on your system.

3. Copy the `FileClassifierJiniService.java` file to the Tomcat `webapps/axis` directory as `FileClassifierJiniService.jws` , changing the file extension.
4. Copy the Jini libraries`jsk-lib.jar` and `jsk-platform.jar` to the Tomcat `webapps/axis/WEB-INF/lib` directory.
5. Start Tomcat *without* a security manager (by default it starts without one).
6. Start a Jini lookup service and any Jini implementation of the `FileClassifier` interface that has been given in this book.
7. Run the Web Service client `ws.TestWS2JiniClient` .

This procedure should run the Web Service, which finds the Jini service, makes a call on it, and returns the result to the Web Service client.

Summary

Bridging between different types of services is not an easy matter, as there are many complex issues to be considered. Nevertheless, it is possible to build upon work performed to build Java Web Services and their Java clients, allowing you to use Jini services and clients in a relatively simple manner.