JMS URL Syntax for the Apache Axis Project

June, 2003

Authors:

Dave Chappell (chappell@sonicsoftware.com), Sonic Software
Ray Chun (rchun@sonicsoftware.com), Sonic Software
Jaime Meritt (jmeritt@sonicsoftware.com), Sonic Software

Version History
This document is an update to the JMS URL Syntax proposal originally submitted to the Axis community in November of 2002.

Acknowledgements
This specification includes feedback from the following individuals:

Phil Adams, IBM
James Snell, IBM
Glen Daniels, Macromedia

Summary

This is a proposal to define a URL syntax for use by the JMS transport layer in Axis. The JMS URL syntax will allow a vendor-neutral way of specifying JMS-specific details, such as message priority and destination, without having to code such details explicitly into the client application, or pass them from a command line.

The benefit to this approach is that with JMS transport details fully contained in a URL, it is possible to switch between transports simply by modifying the transport URL. This simplifies the task of creating JMS client applications and stub classes, and is a necessary first step in adding JMS transport support to Axis' wsdl2java tool.

1. Background

When one uses the JMS transport layer, details of the transport are hidden underneath the client API of Axis. Axis implements the JAX-RPC API. JAX-RPC provides multiple models for constructing both static and dynamic invocations of a service. There are two flavors of dynamic invocation using the Call and Stub interface, and a static model using WSDL2Java stub generation. Both the Call and Stub interfaces allow the use of application-specific properties that get passed down to the handlers, including the transport handler.

Currently the model in Axis 1.0 is to specify the JMS-specific properties from the command line, or in a properties file. The client application then obtains these and places the JMS-specific properties into the Call object by calling the setProperty() method. The

properties are then passed on to the transport handler, where they are extracted and used to perform the appropriate actions in JMS nomenclature. Examples of JMS-specific properties include the location of a ConnectionFactory object, the name of a JMS destination, the type of destination (topic or queue), username and password, and message delivery options such as persistence.

1.1 Programming Model for the JMS Transport in Axis 1.0

The following sample code demonstrates the invocation of a stock quote web service using the JMSTransport class in Axis 1.0.

While the current model works well, the ease of plugging in the JMS transport can be made much more tenable by being able to specify everything in a URL encoding. The goal is to eliminate the need for JMS-specific code in the client application by encapsulating transport logic within the transport implementation.

Example: Sample code using Axis 1.0 JMSTransport

```
Service  service = new Service(new XMLStringProvider(wsdd));

// create the JMS transport
JMSTransport transport = new JMSTransport(connectorMap, cfMap);

// create a new Call object
Call     call    = (Call) service.createCall();

// populate the call object.  In this case it's a method
// invocation with a description of the parameter list and
// the return type.
call.setOperationName( new QName("urn:xmltoday-delayed-quotes",
                                 "getQuote") );
call.addParameter( "symbol",
                   XMLType.XSD_STRING,
                   ParameterMode.IN );
call.setReturnType( XMLType.XSD_FLOAT );

// set the transport object
call.setTransport(transport);

// set some properties.  These will get passed down into the handlers,
// including the transport handler.
call.setUsername(username );
call.setPassword(password );
call.setProperty(JMSConstants.WAIT_FOR_RESPONSE, Boolean.FALSE);
call.setProperty(JMSConstants.PRIORITY, new Integer(5));
call.setProperty(JMSConstants.DELIVERY_MODE,
                 new Integer(javax.jms.DeliveryMode.PERSISTENT));
call.setProperty(JMSConstants.TIME_TO_LIVE, new Long(20000));
```

```
call.setProperty(JMSConstants.DESTINATION, destination);
...

// invoke the call
res = (Float) call.invoke(...);

// shutdown the transport
transport.shutdown();
```

2. URL Syntax

Rather than inventing a rigid syntax, it is desirable to use a URL-encoded query string in accordance with RFC-1738. The intent is to specify a minimal syntax, allowing for the most flexibility for different JMS providers.

Proposed syntax:
jms:/<destination>?[<property>=<value>&]*
The JMS transport implementation will also support the use of ';' as the query delimiter, in addition to '&'.

Note that when specifying the JMS URL in a wsdl file, the ampersand should be encoded as "&" (or one of the alternate delimiters used):

```
…
  <service name="OnewayService">
    <port name="Oneway" binding="tns:OnewayBinding">
      <soap:address
         location="jms:/SampleQ1?vendor=sonicmq&amp;priority=5"/>
    </port>
  </service>
</definitions>
```

The decision to encode the domain as an optional query property is driven by the JMS 1.1 specification and its stated goal of domain unification. Once JMS 1.1 is widely adopted, it is expected that the domain will no longer be required by most vendors.

The encoding of properties in the query string is vendor-specific. A standard set of property names is defined in this document, but vendors are free to use their own encodings. The only restriction is that vendors should not overload any of the reserved property names.

2.1 JMS Send Properties

In the short term, the scope of what can be set in the query-string is limited to the perspective of a JMS sender. The following table illustrates the possible settings in the query string.

Property	Example Value	Comments
vendor	"sonic"	Optional. Defaults to JNDI.
host	"localhost"	
port	"2506"	
domain	"queue", "topic"	Indicates the type of connection factory and destination
replyToDestination	"ReplyToQueue"	May be JNDI lookup name or direct destination name
deliveryMode	"persistent", "non-persistent", "discardable"	
ttl	"10000"	Time-to-live, in ms
priority	0-9	Message priority
waitForResponse	"yes"	Can be used in conjunction with call.setTimeout()?

Again, each vendor may opt not to use the above properties. For instance, a broker connection URL may be encoded using "brokerURL=tcp://localhost:2506" instead of "host=localhost&port=2506".

Note that username and password are not standard query properties. Client applications should set these properties directly on the call object, via call.setUsername(String user) and call.setPassword(String password), as is done when using Http for transport.

2.1.1 JNDI Properties

If the vendor property is omitted, the JMSTransport will assume that secondary lookup should be performed using JNDI. The following properties apply to JNDI only:

Property	Example Value	Comments
initialContextFactory	"com.sun.jndi.fscontext.RefFSContextFactory" "com.sun.jndi.ldap.LdapCtxFactory"	Optional.
jndiProviderURL	"file:///JNDIStore" "ldap://localhost:123"	Optional.
connectionFactory	"MyCF", "com.JMSVendor.TopicConnectionFactory"	Required. May be JNDI lookup name or fully qualified class name

2.2 Examples

Using Sonic:
jms:/SampleQ1?vendor=sonic&domain=queue&brokerUrl=localhost:2506&…
jms:/SampleQ1?vendor=sonic;domain=queue;host=localhost;port=2506;…

Using JNDI:
jms:/MyQ?initialContextFactory=icf &
jndiProviderURL=tcp://ldap.somewhere.com &
queueConnectionFactory=MyQCF &
jmsPriority=5 & …
(note: spaces added for readability)

The connection factory and destination are string properties that may map to a JNDI lookup name. If the JMS provider supports direct instantiation of a CF class, or a named destination, then that is allowable as well.

3 JMS URL Programming Model

With the proposed model, calls to control the JMSTransport object's lifecycle are no longer necessary. In particular, note the absence of calls to construct and shutdown the JMSTransport.

Properties that appear in the query string may be overridden by explicitly setting the property via call.setProperty(..). For instance, the username and password, though optional parameters in the query string, will likely be specified using the call object's setUsername() and setPassword() methods.

Example: JMS URL-based Service Invocation

```
Service  service = new Service(new XMLStringProvider(wsdd));

// create a new Call object
Call     call    = (Call) service.createCall();

// populate the call object.  In this case it's a method invocation with
// a description of the parameter list and the return type
call.setOperationName( new QName("urn:xmltoday-delayed-quotes",
                                 "getQuote") );

call.addParameter( "symbol",
                   XMLType.XSD_STRING,
                   ParameterMode.IN );
call.setReturnType( XMLType.XSD_FLOAT );

// set the jms url
```

```
String sampleJmsUrl = “jms:/SampleQ1” +
                      “&vendor=sonic” +
                      “&deliverymode=persistent” +
                      “&priority=5” +
                      “&replyToDestination=SampleQ2”;
call.setTargetEndpointAddress(new java.net.URL(sampleJmsUrl));

// setting properties explicitly is still supported
// these override properties specified in the URL
call.setProperty(JMSConstants.TIME_TO_LIVE, new Long(20000));
call.setProperty(JMSConstants.REPLY_DESTINATION, “ReplyToQ”);
//…

call.setUsername(username );
call.setPassword(password );

// invoke the call
res = (Float) call.invoke(...);
```

4 Outstanding Issues

Redelivery
How does one check the JMSRedelivered property on receipt of a message? JMS properties table in call object?

5 Futures

5.1 Listener Properties

In the not-too-distant future, we will have asynchronous callback support and client notification in the client engine and the client API. When this is enabled, we will need to be able to specify a listener and listener traits in the query-string. This is likely to change as we evolve the API.

Appendix A: Implementation Strategy

The JMS URL implementation is expected to cover two phases. The first phase will add support for the JMS URL in the transport layer, as described in this proposal. Implementation details are provided below. The second phase will include changes to the wsdl2java tool, for auto-generation of stubs containing JMS URL endpoints.

5.1 Proposed Modifications

The following modifications are required in support of the JMS URL functionality:

1. Registration of the JMS transport class with Axis

   There is currently no Axis transport registered to handle the "jms" protocol. The reason JMS support is possible right now is that the JMSTransport object is explicitly created and set on the Call by the client application, so that the transport lookup is bypassed.

   For JMS URL support, however, the transport must be registered using the static org.apache.axis.client.Call.setTransportForProtocol() method.

2. Registration of the JMS vendor adapter classes

   In the current Axis JMS implementation, the JMSTransport instance is tied to a single JMS vendor implementation (the JMSVendorAdapterFactory will not create adapters from multiple JMS providers). Once the transport is created, all messages sent over that transport are handled by the same provider.

   In order to support JMS URLs from multiple vendors using the same JMSTransport, the JMSVendorAdapterFactory will be enhanced to maintain a list of vendor adapters, keyed on vendor uri.

3. Creation of an Axis message context with JMS URL properties

   The Axis MessageContext is currently created with properties set on the Call object by the client application. This will not change. However, the JMSTransport.setupMessageContextImpl() method will be modified to also extract properties from the JMS URL (with help from the vendor adapters).

   A base JMS adapter will be provided for parsing the standard JMS properties from the URL. Vendors will just need to extend this base class and provide additional code for extracting vendor-specific properties, if any, from the URL.

4. Shutting down the JMS transport

   Once a JMS URL syntax is supported in Axis, the call to explicitly set the JMSTransport is no longer required in the client application, as the appropriate transport will be selected based on the protocol specified in the endpoint address.

Axis handles the creation of transport objects (and associated JMS connections) without requiring direct instantiation of transports by the client application.

There is, however, still a need for the client application to signal transport shutdown (for instance, to close open JMS connections). The proposed mechanism for signaling shutdown of the JMS sender is to set the appropriate property (e.g., QUIT_REQUESTED) in the Call object. Upon receiving a message with this property set to true, the JMS transport will be shut down.