GT3 SOAP Usage Notes

By: Thomas Sandholm 7/14/2003

1 Aim

The aim of this document is to clarify the reasoning behind the use of different SOAP packaging and encoding styles in GT3. Basic WSDL, SOAP, XML Schema and OGSI knowledge is assumed.

2 Design Goals

There are two overall goals when encoding SOAP requests 1) it should be possible to pass any types, determined at run time and defined in an XML Schema specification, inside of a SOAP message, 2) intermediary servers should not have to have any type knowledge at compile time to forward these types inside of a message. These goals are implied by the design of OGSI.

An additional goal, which is more of a programming model issue, is to leverage the functionality of SOAP toolkits, while keeping the application shielded from the details of what is actually sent on the wire: the application should not have to perform any direct XML manipulation or parsing. That is, parameter and operation deserialization and dispatching should ideally be done entirely by the SOAP toolkit.

3 Use Case

Below is a typical use case of what we want to be able to do with a SOAP Engine toolkit.

1. A service is described using WSDL with operations containing a generic message that has parts which can be used as generic placeholders to pass arbitrary types (the idea behind OGSI extensibility elements)
2. A set of XML Schema complex types are specified separately
3. A wsdl compiler is used to provide client side stub code from WSDL generated in step 1
4. An XML Schema compiler is used to generate programming language specific types from the definitions in step 2

5. The hosting environment is configured to map the types generated in step 4 to hosting environment specific XML serializers and deserializers
6. A client makes a call using a stub generated in step 3, and passes in a type generated in step 4
7. The server gets the call and if a mapping was provided for the inputs in step 5, a type from step 4 is created and populated. If no mapping is found an XML Infoset (e.g. W3C DOM Element) is presented to the application.

4 Requirements

1. **It must be possible to pass arbitrary XML Schema defined types as parameters in an operation/parameter wrapped request**

Either xsd:anyType or xsd:any namespace="##other" may be used for this purpose. If the namespace of a particular type is used, then xsd:any namespace="<type namespace>" must be used. The operation names must be serialized in the SOAP messages.

2. **The interface of the stub must not reveal anything about the binding or the particular encoding used**

The name and the signature of the interface should not reveal more information than is known from the portType definition. Whether some requests use SOAP encoding and others literal should be hidden from the client.

3. **The server side implementation interface must not reveal anything about the binding or the particular encoding used. Ideally this interface would be the same as for the client side stub within the same hosting environment**

Similar to requirement 2, the interface a server side programmer has to implement should not expose what SOAP encoding techniques was used by the SOAP engine. Generic types from requirement 1 should be exposed as a generic programming language type (like java.lang.Object) that can either be cast to an explicit type or an XML Infoset representation if no mapping was found. See requirement 5.

4. **It must be possible to configure the hosting environment to map XML Schema namespaces to types handling serialization/deserialization on behalf of the user – unless this can be done automatically at run time.**

It is not enough if the generated client stubs know how to serialize a request; new types may be introduced at runtime, so there must be a flexible way of making the SOAP engine aware of these types.

5. **If a mapping is not configured, or if not enough run time information exists to determine a specific mapping it should be possible to set a default mapping to use a generic XML Infoset type**

If a receiver of a request has no means of deserializing a parameter into a reasonable programming language type for the hosting environment, the hosting environment should provide an object with an API allowing the programmer to browse through the XML Infoset.

6. **There should be an API that allows easy transformations between XML Infoset representations and Programming language objects**

   This is important in order to allow XML based queries to do arbitrary searches over a set of XML fragments without exposing the XML to the application.

5 WSDL

The RPC/literal model seems to be closest, conceptually, to the requirements just listed. Due to the vague definition of RPC/literal in WSDL 1.1, not many toolkits implement it, though. The document/literal implementation in Microsoft .NET is, however, also a very close match to our requirements, since it does the RPC-like operation and parameter wrapping we need. In order to interoperate with .NET, a number of other toolkits like Apache Axis also support this model, and call it the wrapped model. From a pure WSDL point of view it is just marked as document literal with some conventions on how to do the wrapping. All messages will contain only one message part, which is an XML Schema complex type with the same name as the operation. The complex type further wraps all the parameters in a sequence of elements. By basing our message encoding and wrapping on the more flexible document/literal model in WSDL it is easy to allow for deviations and changes in these conventions as the toolkits evolve.

Example of wrapped WSDL:

```
<types>
…
<xsd:element name="add">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="value" type="xsd:int"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
…
<message name="AddInputMessage">
  <part name="parameters" element="tns:add"/>
</message>
…
```

6 SOAP

A SOAP message Body will be encoded as follows:

```
<Body>
  <operationNamespace:operation>
    <operationNamespace:parameterName>
      <typeNamespace:type>…
```

The operation and the operationNamespace are taken from the WSDL specification, and can be hard coded in the generated stub. The typeNamespace and the type could be determined at runtime, depending on what type the client passes in.

7 OGSI

There are at least five scenarios in OGSI where a flexible type-encoding model as discussed above is needed:

1. The most typical use case would be to pass around service data elements returned by findServiceData queries.
2. Query expression languages should be well defined in XML Schema and there must be an easy way for clients to construct a query expression without using low level XML APIs.
3. WSDL documents are frequently passed across the wire, this is a case when we just want a mapping to an XML Infoset.
4. In Notification Sink notifications we want to be able to specify the format of the notification message in XML Schema.
5. When publishing a service to a registry (ServiceGroupRegistration) we want to push arbitrary service information, which can later be queried, into the registry.

8 Axis

The current GT3 Core Java implementation built on top of Apache Axis fulfills all the 6 requirements listed in section 6 above. It is based on the standard JAX-RPC type mapping and serialization framework. An xsd:any (XML Schema any element) definition maps to an Axis AnyContent Bean that allows you to view the XML payload as a list of generic JAX-RPC SOAPElements (with Axis extensions). Since the xsd:any mapping to Java is not standardized in JAX-RPC yet, we provide an AnyHelper API in GT3 Core that simplifies the usage of xsd:any types as well as shields the user from Axis specific extensions.
The actual serialization engine underlying the AnyHelper is JAX-RPC based, which makes it easy to use in conjunction with SOAP messaging. The helper has three basic features: it can convert an xsd:any payload into 1) a DOM Element, 2) a Java type, or 3) a String. Availability of type mappings and application context will determine which conversion that is to be performed.

Although all the OGSI messages are sent with the wrapped literal encoding model described in this document, we allow other Grid service operations to use the RPC/encoded model. This may be useful in scenarios where WSDL is generated from a Java interface, or where a legacy WSDL interface is to be transformed into a Grid service interface. As a rule of thumb, if no extensibility elements like xsd:any are needed the RPC/encoded model could be used for the SOAP payload as well. Using RPC/encoded, however, implies that no custom attributes can be used, which is why we cannot use it for something like OGSI service data. Specifying RPC/encoded vs Wrapped

(document/literal) must be done on an operation level in WSDL, but a portType definition may mix the two models.

9 Future

Although the current encoding model and conventions satisfy both the flexibility requirements of OGSI, and leverages the SOAP Toolkit dispatching mechanisms, we might want to extend this model in the future to allow more relaxed document literal payload mappings to operations. This would suffer from not being able to map so easily into an RPC dispatcher, but could be useful for workflow engine dispatchers. The document/literal model is generic enough to allow any encoding models to be applied on top of it, which our current usage of RPC/encoded shows. When other standard encoding models appear we hope to accommodate them too in a similar way.

There is one limitation of encoding XML Schema types over SOAP that schema designers should be aware of. When serializing simple types as XML attributes, there is no standard way of marking up the payload with the target types, hence the deserializer needs to have out of band access to WSDL meta data to deserialize the type correctly. Another unfortunate side effect of this is that the attribute can not be easily deserialized if its type is not known until run time, e.g. when using polymorphism or union types, in which case the final deserialization decisions have to be made by the application.