Client-side SOAP in S

Duncan Temple Lang
Bell Labs
duncan@research.bell-labs.com

December 4, 2005

Abstract

This gives a brief introduction to the client-side SOAP mechanism provided by the SSOAP package. SOAP stands for the Simple Object Access Protocol and is a means for making remote functions calls in a client-server architecture using XML and HTTP. This is one of the primary Web Services vehicles. This package leverages the XML package for parsing and generating XML from within S and the HTTP connection classes to send and receive the communications. The package provides a mechanism to automatically generate S code that implements an interface to a Web server described by a WSDL (Web Service Description Language) document.

The SSOAP package is currently a basic attempt to implement a client-side mechanism for invoking SOAP methods. SOAP stands for the Simple Object Access Protocol and is documented in [?]. It uses XML to encode remote procedure calls (RPC) from a client to a server and transmit the result back to the client. While XML is used to describe the content of the call and result, HTTP is used to implement the communication of these data. In S, we have the XML package with which we can read and write XML. R provides the *socketConnection()* function with which we can create a read-write stream between R and a SOAP server. Combining each of these with the *readLines()*, we can implement the entire client-server communication mechanism in S code and implement basic calls to SOAP servers.

The package is at a very early stage and is more of an experiment and example of using XML. There are several details to work out before it is functional, and even more before it is robust. The hope is to encourage others to investigate it and bring their knowledge of SOAP, etc. to help fix the problems.

July 20th, 2003. Recent revisions have made the package more robust and added features to "compile" client-side functions for server methods that include information about parameter and return types. Thanks go to Vincent Carey, Robert Gentleman and Jianhua Zhang all of Harvard and the BioConductor project for illustrating issues.

The top-level function for invoking a SOAP method is *.SOAP()*. There are several pieces that one has to pass to this to define the SOAP method request. The first thing you have to provide is the identity of the SOAP server. You create a SOAP server object in S using the *SOAPServer()* function and you give it the name of the host machine (e.g. services.ensembl.org) and the file within the server that identifies the SOAP server. If the server is not listening for requests on the usual HTTP server port (i.e. 80), you can also specify the port on which it is listening. For example, to communicate with the Ensembl soap server, we can specify the server as

?? ⟨ ??⟩≡

```
SOAPServer("services.ensembl.org", "cgi-bin/ensembl_rpcrouter", 7070)
```

Having identified, the server, we must also specify the name of the method to invoke and provide any arguments that are needed to parameterize the call to the method. The name of the method is given as a simple string via the *method* argument (what a surprise!). The arguments are given via S's ... mechanism as regular S objects.In some calls, you will know the names of the parameters for the SOAP method. In that case, you can give them in the usual S fashion:

?? ⟨ ??⟩+≡
```
.SOAP(server, "myMethod", country1 = "England", country2 = "Japan")
```

These names are used XML tags within the SOAP request to identify the arguments.

?? ⟨ ??⟩+≡
```
<SOAP-ENV:Body>
  <x:getRate xmlns=...>
     <country1 xsi:type="xsd:string">England</country1>
     <country2 xsi:type="xsd:string">Japan</country1>
  </getRate>
</SOAP-ENV:Body>
```

In other cases, you will only know the order of the parameters and you need not provide names. The *.SOAP()* will use "made-up" names for the XML elements.
Most SOAP servers will also require that you specify what is called a *SOAPAction* value. This is a string that is included in the HTTP request to the server to help it interpret the request. How you find this string is server-dependent. However, once you have it, you specify it in the *.SOAP()* call via the *action* argument.

For some servers, the actual value is the one announced in the server's documentation combined with the name of the method. For example, a SOAPAction of `urn:xmethods-delayed-quote` for a method getQuote would be transformed to `urn:xmethods-delayed-quote#getQuote`. Obviously, it is convenient and less error-prone to have the *.SOAP()* function do this for us rather than have to input the same information in two places. So, by default, this is what the *.SOAP()* function does using its *handlers()* argument. Specifically, this is a collection of functions that are used to modify the inputs and process the outputs for the basic *.SOAP()* mechanism. They allow us to parameterize the *.SOAP()* function without having to write methods for standard data types or override functions. We can supply very specific functions that take inputs from the *.SOAP()* call and return suitably modified values. The default handlers are available by calling *SOAPHandlers()*. And within this, the *action* element is is responsible for pasting the user-supplied *action* value with the method name. If this is not what is expected for the particular server, you can specify a different collection of *handlers*.

In addition to the `SOAPAction` value, the server also expects a particular XML namespace to be used for the XML tags that give the method name and the arguments. This can be given as a simple string via the *xmlns* argument. If you explicitly want to control the local identifier used in the tags (e.g. the `x` in *x:methodName*) for this namespace, you can supply that name in the character vector using

?? ⟨ ??⟩+≡
```
xmlns = c(x="http://myNamespace/URI")
```

More than one namespace can be used, but while the others will be included in the XML for the request, they will typically play no role in the invocation.
The final aspect of the *.SOAP()* call is the *nameSpaces* argument. The SOAP request is enclosed within an *Envelope* and within that an *Body* element within the XML. These are qualified with suitable namespace identifiers and declarations to denote that they are SOAP requests and to specify the version of SOAP and how they and the sub-nodes should be interpreted. So we need to add these namespace declarations to the XML *Envelope* element in the request. The *nameSpaces* argument used to identify these "global" or SOAP-level names by giving the name-value pairs as a named character vector. Each name-value pair gives the identifier for the namespace and its URI.

In much the same way that we may want to omit or override some of the default namespaces, we call a function *SOAPNameSpaces()* to get the vector namespaces. By default, this returns the usual collection. One can switch between different versions of namespaces such as version 1.1 and 1.2 by giving the appropriate name as the value of the *version* parameter. This is used to index the list *.SOAPDefaultNames* and indeed, you can add additional collections to that list. Regardless of which collection of namespaces you select in this list, you can also specify your own values for particular elements within that collection or simply augment it with new ones. We do this by listing the name-value pairs in the call to *SOAPNameSpaces()*.

If you want to get a subset of the values within a particular collection of namespaces, you can give the names of the elements to include or, if more convenient, the ones to exclude. For example, the following two commands give the same collection, but work by including the elements of interest and excluding the extraneous ones, respectively.

?? ⟨ ??⟩+≡
```
SOAPNameSpaces(include = c("SOAP-ENC", "SOAP-ENV"))
SOAPNameSpaces(exclude = c("xsi", "xsd"))
```

The main purpose of the *SOAPNameSpaces()* function is to make it easy to control what is include in a single call. It avoid the need to create temporary variables with the correct values, pass them in the call and then remove them. The function approach allows the collection to be specified in-line within the call.

1 Converting the S values to SOAP

The basic mechanism for converting S objects to SOAP is implemented in the functions *toSOAP()*, *toSOAPArray()* and *toSOAPStruct()*. These can be used in converting arguments to a SOAP call or implementing a SOAP server and converting the result to a SOAP object. The mechanism is simple. Primitive scalars are mapped to their equivalents in the XSI/XSD schema. Real numbers map to S numeric types, booleans map to logical, int to integers and strings to character vectors. Non-scalar primitives (i.e. vectors of length > 1) and unnamed lists are mapped to SOAP arrays. Named lists are mapped to SOAP structures.

2 The Result

Typically we will be interested in the result of the *.SOAP()* call. This is procssed when the server returns it via another SOAP envelope. Assuming there were no errors, we extract the contents of that envelope and pass the XML node to *fromSOAP()*. This attempts to convert it to an S object based on some basic ideas. It handles the primitive types defined in the XSI and XSD schema in the natural manner. Real numbers map to S numeric types, booleans map to logical, int to integers and strings to character vectors. Arrays in SOAP are mapped to lists in S. If they have the same primitive type, we can collapse them to a regular vector. (This is not currently done?). Support for partially transmitted and sparse SOAP arrays is implemented, but multi-dimensional arrays have yet to be enabled. (Please let me know of an example.) SOAP complex objects are mapped to S3-style objects in which we create a list with named elements taken from the sub-nodes of the XML result. This works recursively and the same rules apply to these sub-nodes.

If you want to take over the conversion, you can specify a different function in the *SOAPHandlers()*. It should be named *result* and should take 4 arguments. These are

a the full XML content returned by the server, i.e. the *Envelope* element

b the HTTP header given as a vector of name-value pairs. This can be useful for determining auxillary information about the XML.

c the name of the method that was invoked. This may be needed to get the appropriate element in the XML tree that contains the result.

d the SOAP server object identifying which server was used. If this has specific characteristics about how it returns values, we may want to build this into the handler.

3 Errors

If there was an error in the SOAP call, the server responds with an HTTP error. Along with the header information, the server also adds some information about the source of the error. We return this as a SOAPFault object that identifies the nature of the failure and contains any extra information provided by the server. Essentially, these are exceptions that we would throw if we had an exception system.

There are 4 types of built-in SOAP errors and they each have the same structure inherited from SOAP-Fault.

SOAPVersionMismatchFault

SOAPMustUnderstandFault

SOAPClientFault

SOAPServerFault

If an error is not one of these types, we create a SOAPGeneralFault object and include its *faultcode* as a slot in the object.

4 Examples

?? ⟨ ??⟩+≡

```
.SOAP(SOAPServer("services.xmethods.net", "soap"),
       "getRate", country1="England", country2 = "Japan",
       action="urn:xmethods-CurrencyExchange")

.SOAP(SOAPServer("services.xmethods.net", "soap/servlet/rpcrouter"),
      "getPrice", "0596000278",
      action="urn:xmethods-BNPriceCheck")

s <- SOAPServer("http://services.xmethods.net/soap")
.SOAP(s,
      "getQuote", "AMZN",
      action="urn:xmethods-delayed-quotes#getQuote")


.SOAP(SOAPServer("services.soaplite.com", "temper.cgi"),
       "c2f", 37.5,
      action="http://www.soaplite.com/Temperatures")


 # Different action and namespace.
 # Specify handlers=NULL to avoid the additional processing of the
 # SOAPAction string, i.e. the appending of the method name.
 # This kills off all the handlers. If we want to remove only the
 # "action" element, we have to be more explicit.

s1 <- SOAPServer("services.soaplite.com", "interop.cgi")
.SOAP(s1,
      "echoString", "From R",
      action="urn:soapinterop",
      xmlns=c(namesp1="http://soapinterop.org/"),
      handlers =NULL)
```

5 Service Declarations

Somtimes, a server's methods are published in a WSDL [?] (Web Services Description Language?) file. This is an XML description of the different methods and the details of how to call them. It lists the expected types of the arguments and return value, the namespaces and SOAP action strings, etc. Since this is an XML file, we can easily read it into S and use the information to generate function definitions that implement the *.SOAP()* calls. This is very similar to reading IDL files in CORBA and DCOM to generate client bindings.

July 2003 In response to a query by Vincent Carey, we have now implemented a basic mechanism to read WSDL files and create S functions that can be used to call SOAP server methods. This is intended to be done by a person interested in a particular SOAP server and who wants to create a package providing access to it from R. In other words, this is a pre-processing step that is done once and not by each user and each time she wants to access the SOAP server.

The idea is quite simple. We start by reading the WSDL file into R using the simple DOM parser from the `XML` package. The function *processWSDL()* does this and returns an object of class SOAPServerDescription. This contains information about how to connect to the server, its methods and also any data types it defines for use in these methods. *It is possible for multiple servers to be defined in a single WSDL file. We have left this possibility open but have not been entirely consistent. Much of the testing and development has been done using the KEGG WSDL file.*

?? ⟨ ??⟩+≡
```
library(SSOAP)
def = processWSDL()
```

Given this definition, we can now use the *genSOAPClientInterface()* function to generate S functions for each of the methods in the server.

?? ⟨ ??⟩+≡
```
ff = genSOAPClientInterface(def = tmp, tmp@name)
```

This has the additional side-effect of defining S4-style classes for each of the complex compound data types defined in the WSDL file. It does this by iterating over the slots described there and adding them as slots to an S4 class of the same name as the compound complex type. For example, the `GENESResult` type in the `KEGG.wsdl` file

?? ⟨ ??⟩+≡
```
        <xsd:complexType name="GENESResult">
                <xsd:all>
                        <xsd:element name="kid" type="xsd:string"/>
                        <xsd:element name="keggdef" type="xsd:string"/>
                </xsd:all>
        </xsd:complexType>
```

gives rise to a corresponding S4 class

?? ⟨ ??⟩+≡
```
> getClass("GENESResult")

Slots:

Name:        kid   keggdef
Class: character character
```

Let's return to the methods generated by the call to *genSOAPClientInterface()*. There is a function for each of the operation elements in the bindings element of the WSDL document. We map the names from SOAP style to S4 style. By this, we mean that we remove any underscores (_), capitalize the first letter of all but the resulting first word, and then collapse the words into a single word by removing the space. For example, $\texttt{get}_a\mathit{ll}_n\mathit{eighbors}_b\mathit{y}_g\mathit{eneismappedto}\texttt{getAllNeighborsByGene}.\mathit{Thisisacommonprogrammingconventionandavoidscomplexityi}$

Each of these functions has access to the S4-WSDLMethod object that was created when from the WSDL file description. This object contains all the information about the method including its name, parameter types, return value type and SOAP action. (At present, we use lexical scoping to make this object available to the function. In the future we could perform direct substitution when creating the function, but this is harder than it need be at present.) The body of this function simply involves a call to the *.SOAP()* function, providing it with the server, arguments, type information, SOAP action, etc. These functions have a formal list of parameters that are computed from the WSDL description and pass these arguments in a call via the ... parameter of the *.SOAP()* function. In this way, the functions provide more information for the caller than a regular SOAP-call. Additionally, the function passes the type information about the SOAP parameters and these are used to control how the arguments are converted to their SOAP equivalent. This is important and convenient as it provides a way for the user to pass a scalar value, for example, and have it be represented appropriately as a scalar or an array. This removes the ambiguity introduced from S's unusual but consistent view of scalars being simply vectors of length 1.

6 Other SOAP Tools

Other SOAP interfaces include SOAP::Lite, Apache's Java SOAP classes.