A Young Person's Guide to The Simple Object Access Protocol: SOAP Increases Interoperability Across Platforms and Languages

Don Box

This article assumes you're familiar with COM and XML

Level of Difficulty 1 2 3

Code for this article: SOAP.exe (32KB)

SUMMARY The Simple Object Access Protocol (SOAP) facilitates interoperability among a wide range of programs and platforms, making existing applications accessible to a broader range of users. SOAP combines the proven Web tech-no-logy of HTTP with the flexibility and extensibility of XML.

This article takes you on a comprehensive tour of Object RPC technology to help you understand the foundations of SOAP and the ways it overcomes many of the limitations of existing technologies, including DCOM and CORBA. This is followed by a detailed treatment of the SOAP encoding rules with a focus on how SOAP maps onto existing ORPC concepts.

hen I began my computing career in 1984, most programmers didn't care about network protocols. r, sometime in the 1990s networking became ubiquitous, and now it's hard to imagine using a computer some form of connectivity. Today the average programmer is more interested in building scalable, ted applications than implementing floating, semi-transparent, nonrectangular, owner-drawn Coolbars in MFC.

Programmers prefer to think in terms of programming models, not network protocols. Though that's generally a good thing, in this article I'll discuss the Simple Object Access Protocol (SOAP), a network protocol that happens to have no explicit programming model. This doesn't mean that the architects of SOAP (including the author) are out to fundamentally change the way you program. Rather, one of the primary goals of SOAP is to make your existing programs more accessible to a broader range of users. To this end, there is no SOAP API or SOAP Object Request Broker (ORB). Instead, SOAP assumes that you will use as much existing technology as possible. Several major CORBA vendors have committed to support the SOAP protocol in their ORB products. Microsoft has committed to support SOAP in future versions of COM. DevelopMentor has developed reference implementations that make SOAP accessible to any Java-language or Perl programmer on any platform.

The guiding principle behind SOAP is to "first invent no new technology." SOAP uses two existing and widely deployed protocols: HTTP and XML. HTTP is SOAP's RPC-style transport, and XML is its encoding scheme. With a few lines of code and an XML parser, HTTP servers such as Microsoft® Internet Information Server (IIS) and Apache instantly become SOAP ORBs. Given the fact that over half of the planet's Web traffic is directed at IIS or Apache, SOAP benefits from the proven engineering and wide availability of these two products. This does not mean, however, that all SOAP requests must be routed through a Web server. Traditional Web servers are just one way to dispatch SOAP requests. Web services like IIS or Apache are sufficient, but by no means necessary for building SOAP-enabled applications.

As this article will describe, SOAP simply codifies the use of XML as an HTTP payload. The most common application of SOAP is as a Remote Procedure Call (RPC) protocol. To understand how SOAP works, it is useful to take a brief look into the history of RPC protocols.

RPCs Throughout History

The two dominant communication models for building distributed applications are message passing (often combined with queuing) and request/response. Message passing systems typically allow any party to send messages at any time. Request/response protocols restrict the communication pattern to request/response pairs. Messaging-based applications are acutely aware that they are communicating with external concurrent processes and require an

explicit design style. Request/response-based applications more closely resemble a single-process application, since the application that sends the request is more or less blocked until it receives the response from the second process. This makes request/response communications a natural fit for RPC applications.

While both messaging and request/response have their advantages, either one can be implemented in terms of the other. Messaging systems can be built using lower-level request/response protocols. For example, Microsoft Message Queue Server (MSMQ) uses DCE RPC internally for most of its control logic. RPC systems can be built using lower-level messaging systems. MSMQ provides a correlation ID for exactly this purpose. For better or worse, most applications tend to use RPC protocols due to their wider availability, simpler design, and natural mapping to traditional programming techniques.

During the 1980s, the two dominant RPC protocols were Sun RPC and DCE RPC. The most popular Sun RPC application is the Network File System (NFS) used by most Unix systems. The most popular DCE RPC application is Windows NT®, which uses the DCE RPC protocol for a number of system services. Both of these protocols proved to be quite functional and adaptable to a wide range of applications. However, as the decade neared an end, the industry's obsession with object-orientation went into full swing, motivating programmers across the globe to forge a marriage between object-oriented languages and RPC-based communications.

The 1990s brought Object RPC (ORPC) protocols that attempted to marry object orientation and network protocols. The primary difference between ORPC and the RPC protocols that preceded them was that ORPC codified the mapping of a communication endpoint to a language-level object. Somewhere in the header of each ORPC request

was a cookie that the server-side plumbing could use to locate the target object in the server process. Often this cookie was just an index into an array, but other techniques were (and are) often used, such as using symbolic names as keys into a hash table.

Figure 1 ORPC Request and Response

Figure 1 shows a typical ORPC request and response message. There are several request header components that are used by the server-side plumbing to dispatch the call. The object endpoint ID is used to locate the target object inside the server process. The interface ID and method ID are used to determine which method to call on the target object. The payload is used to transport the values of any [in] and [in, out] parameters as part of the request (or [out] and [in,out] parameters in the case of a response). Note that optional protocol extensions can appear between the header fields and the payload. This is standard practice in protocol design, as it allows new services to be piggybacked on an ORPC request or response. Most ORPC systems use this area for transmitting additional context information (such as transaction information and causality identifiers).

At this time, the two dominant ORPC protocols are DCOM and CORBA's Internet Inter-ORB Protocol (IIOP) flavor of the General Inter-ORB Protocol (GIOP). The request formats from DCOM and IIOP/GIOP are extremely similar, as shown in Figure 2.

Both protocols use an object endpoint ID to identify the target object, as well as a method identifier to determine which method to invoke.

There are two differences between the protocols worth noting. The primary difference between the two protocols is that with IIOP/GIOP, the interface ID is implicit, since a given CORBA object only implements one interface (although the Object Management Group (OMG) is currently standardizing support for multiple interfaces per object). Another subtle difference between DCOM and IIOP/GIOP requests is the format of parameter values in the payload. In DCOM, the payload is written in a format known as Network Data Representation (NDR). In IIOP/GIOP, the payload is written using Common Data Representation (CDR) format. Both NDR and CDR deal with the differing data representations used on various platforms. However, there are some minor differences between these two formats that make them incompatible with one another.

Another key distinction between ORPC and RPC protocols is how communication endpoints are named. In ORPC protocols, some transmissible representation of an ORPC endpoint is needed to communicate object references across the network. In CORBA/IIOP, this representation is called an Interoperable Object Reference (IOR). IORs contain addressing information in a portable format that any CORBA product can resolve to an object endpoint. In DCOM, this representation is called an OBJREF, which combines distributed reference counting with endpoint/object identification. Both CORBA and DCOM provide higher-level mechanisms for finding object endpoints on the network, but at the end of the day these mechanisms all map down to IORs or OBJREFs. Figure 3

shows how an IOR/OBJREF relates to the addressing information found in IIOP/DCOM request messages.

What's Wrong with this Picture?

While DCOM and IIOP are both solid protocols, the industry has not shifted completely to either one. The lack of convergence is partly due to cultural issues. Additionally, the technical applicability of both protocols has been called into question as organizations have tried to standardize on one protocol or the other. The conventional wisdom is that DCOM and CORBA are both reasonable protocols for server-to-server communications. However, both DCOM and IIOP have severe weaknesses for client-to-server communications, especially when the client machines are scattered across the Internet.

DCOM and CORBA/IIOP both rely on single-vendor solutions to use the protocol to maximum advantage. Though both protocols have been implemented on a variety of platforms and products, the reality is that a given deployment needs to use a single-vendor's implementation. In the case of DCOM, this means every machine runs Windows NT. (Although DCOM has been ported to other platforms, it has only achieved broad reach on Windows®.) In the case of CORBA, this means that every machine runs the same ORB product. Yes, it is possible to get two CORBA products to call one another using IIOP. However, many of the higher-level services (such as security and transactions) are not generally interoperable at this time. Additionally, any vendor-specific optimizations for same-machine communications are very unlikely to work unless all applications are built against the same ORB product.

DCOM and CORBA/IIOP both rely on a closely administered environment. The odds of two random computers being able to successfully make DCOM or IIOP calls out of the box are fairly low. This is especially true when security is involved. While it is possible to write a shrink-wrap application that can use DCOM or IIOP successfully, doing so requires much more attention to detail than the typical sockets-based application. This is especially applicable to the unglamorous but necessary task of configuration/installation management.

DCOM and CORBA/IIOP both rely on fairly high-tech runtime environments. While in-process COM is deceptively simple, building the COM/DCOM remoting plumbing is definitely not a weekend project. IIOP is a simpler protocol to implement than DCOM, but both protocols have their fair share of arcane rules dealing with data alignment, type information, and bit twiddling. This makes it difficult for the average programmer to simply cruft up a CORBA or DCOM call without the benefit of an ORB product or OLE32.DLL.

Perhaps the most damning limitation of DCOM and CORBA/IIOP is their inability to work in Internet scenarios. In the case of DCOM, it is unlikely that the average user's Bondi-blue iMac or cheap PC clone running Windows 95 will be able to perform domain-based authentication with your servers. Worse, if a firewall or proxy server separates the client and server machines, the likelihood of either IIOP or DCOM packets getting through is extremely low due to the HTTP bias of most Internet connectivity technology. While vendors like Microsoft, Iona, and Visigenic have all built tunneling technology, these products tend to be very sensitive to configuration mistakes and are not interoperable.

None of these issues impact the use of DCOM or IIOP within a server farm. The number of host machines in a server farm is relatively small (hundreds, not tens of thousands), which marginalizes the cost of DCOM's ping-based lifecycle management. Chances are that all of the host machines in the server farm are under a common administrative domain, which makes consistent configuration quite likely. The relatively small number of machines also helps to keep the costs of using commercial ORB products under control, as a smaller number of ORB licenses are needed. If IIOP is only spoken within the server farm, a smaller number of ORB licenses are needed. Finally, it is likely that all of the host machines in a server farm will have direct IP connectivity, removing the firewall-related problems of DCOM and IIOP.

HTTP as a Better RPC

It is common practice to use DCOM or CORBA within a server farm, but to use HTTP to enter the server farm from a client machine. HTTP is a very RPC-like protocol that is simple, widely deployed, and more likely to function in the face of firewalls than any other protocol known to man. HTTP requests are typically handled by Web server software (such as IIS and Apache), but an increasing number of application server products are supporting HTTP as a native protocol in addition to DCOM and IIOP.

Like DCOM and IIOP, HTTP layers request/response communications over TCP/IP. An HTTP client connects to an HTTP server using TCP. The standard port number used in HTTP is port 80, but any port can be used. After establishing the TCP connection, the client can send an HTTP request message to the server. The server then sends an HTTP response message back to the client after processing the request. Both the request and response messages can contain arbitrary payload information, typically tagged with the Content-Length and Content-Type HTTP headers. The following is a legal HTTP request message:

```
POST /foobar HTTP/1.1
Host: 209.110.197.12
Content-Type: text/plain
```

```
Content-Length: 12

Hello, World
```

You may have noticed that the HTTP headers are just plain text. This makes it easy to diagnose HTTP problems using a packet sniffer or text-based Internet tools like telnet. The text-based nature of HTTP also makes it easily adaptable to low-tech programming environments popular in Web development.

The first line of an HTTP request contains three components: the HTTP method, the Request-URI, and the protocol version. In the previous example, these correspond to POST, /foobar, and HTTP/1.1, respectively. The Internet Engineering Task Force (IETF) has standardized a fixed number of HTTP methods. GET is the HTTP method used to surf the Web. POST is the most commonly used HTTP method for building applications. Unlike GET, POST allows arbitrary data to be sent from the client to the server. The Request-URI (Uniform Resource Identifier) is simply a token used by the HTTP server software to identify the target of the request (much like an IIOP/GIOP object_key or a DCOM IPID). For more information on URIs see the sidebar, "URIs, URLs, and URNs." The protocol version in this example is HTTP/1.1, which indicates that the rules of RFC 2616 are to be observed. HTTP/1.1 added several features to its predecessor (HTTP/1.0), including support for chunked data transfer and explicit support for keeping TCP connections alive across HTTP requests.

The third and fourth lines of the request specify the size and type of the request payload. The Content-Length header specifies the number of bytes of payload information. The Content-Type identifier specifies the syntax of the payload information as a MIME type. HTTP (like DCE) allows the client and server to negotiate the transfer syntax used to encode information. Most DCE applications use NDR. Most Web applications use text/html or other text-based syntaxes.

Pay attention to the blank line between the Content-Length header and the request payload in the code sample. Individual HTTP headers are delimited by a carriage-return/line-feed sequence, and the headers are delimited from the payload using an extra carriage-return/line-feed sequence. The request then contains raw bytes whose syntax and length are identified by the Content-Length and Content-Type HTTP headers. In this example, the content is the 12-byte plain text string "Hello, World".

After processing the request, the HTTP server is expected to send an HTTP response back to the client. The response must contain a status code indicating the outcome of the request. The response can also contain arbitrary payload information much like the request message. The following is an HTTP response message:

```
200 OK
Content-Type: text/plain
Content-Length: 12

dlroW ,olleH
```

In this case, the server returned a status code of 200, which is the standard success code for HTTP. Had the server been unable to decode the request, it would have returned the following response instead of the one shown previously:

```
400 Bad Request
Content-Length: 0
```

Had the HTTP server decided that requests for the target URI should be temporarily redirected to a different URI, the following response would have been returned:

```
307 Temporarily Moved
Location: http://209.110.197.44/foobar
Content-Length: 0
```

This response informs the client that the request could be satisfied by retransmitting it to the endpoint identified in the Location HTTP header.

All of the standardized status codes and headers are documented in RFC 2616. Very few of them relate directly to SOAP users, with one notable exception. In HTTP/1.1, the underlying TCP connection is reused across multiple request/response pairs. The HTTP Connection header allows either the client or the server to close the underlying connection. By adding the following HTTP header to a request or response, both sides are required to shut down their TCP connections after processing the request.

`Connection: close`
To keep the TCP connection alive when interoperating with HTTP/1.0 software, it is recommended that the sender add the following HTTP header to each request or response:

`Connection: Keep-Alive`
This header disabled the default HTTP/1.0 behavior of resetting the TCP connection after each response.
One of the advantages of HTTP is its wide deployment and acceptance.

```
static String http_reverse(String serverAddress, String s) throws Throwable
{
    java.net.URL url = new java.net.URL(serverAddress);
    java.net.HttpURLConnection conn =
        (java.net.HttpURLConnection)url.openConnection();
    conn.setRequestMethod("POST");
    conn.setRequestProperty("Content-Type", "text/plain");
    conn.setDoOutput(true);
    conn.connect();
    java.io.OutputStream out = conn.getOutputStream();
    out.write(s.getBytes());
    out.close();
    java.io.InputStream in = conn.getInputStream();
    byte[] rgb = new byte[1024];
    int cb = in.read(rgb);
    in.close();
    return new String(rgb, 0, cb);
}
```

Figure 4 shows a simple Java-language program that sends the request shown previously and parses out the resultant string from the response. The following is a simple C program that uses CGI to read the string from the HTTP request and write the reversed version back out through the HTTP response.

```
#include <stdio.h>

int main(int argc, char **argv) {
    char buf[4096];
    int cb = read(0, buf, sizeof(buf));
    buf[cb] = 0;
    strrev(buf);
    printf("200 OK\r\n");
    printf("Content-Type: text/plain\r\n");
    printf("Content-Length: %d\r\n", cb);
    printf("\r\n");
    printf(buf);
    return 0;
}

public class ReverseServlet extends javax.servlet.http.HttpServlet
{
    public void doPost(javax.servlet.http.HttpServletRequest
                           request,
                       javax.servlet.http.HttpServletResponse
                           response)
                  throws java.io.IOException
    {
        byte[] rgb = new byte[request.getContentLength()];
        int cb = request.getInputStream().read(rgb);
        StringBuffer sb = new StringBuffer(new String(rgb, 0,
                                                      cb));
        response.getWriter().print(sb.reverse());
    }
}
```

Figure 5 shows a more modern version of the server implemented as a Java-language servlet to avoid the overhead of CGI's process-per-request model.

In general, CGI is the way to write HTTP server code for the lowest common denominator. Virtually every HTTP server product provides a much more efficient mechanism to get your code to process an HTTP request. IIS provides ASP and ISAPI as the native mechanisms for writing HTTP code. Apache allows you to write modules in C or Perl

that run inside the Apache daemon. Most application server products allow you to write Java-language servlets, COM components, EJB session beans, or CORBA servants based on the Portable Object Adapter (POA) interface.

XML as a Better NDR

HTTP is a fairly functional RPC protocol that provides most—if not all—of the functionality of IIOP or DCOM in terms of framing, connection management, and support for serialized object references. (URLs are surprisingly close to IORs and OBJREFs in functionality.) What HTTP lacks is a single standard format for representing the parameters of an RPC call. This is where XML comes in.

Like NDR and CDR, XML is a platform-neutral data representation protocol. XML allows data to be serialized into a transmissible form that is easily decoded on any platform. XML has the following characteristics that differentiate it from NDR and CDR:

- There is a plethora of XML encoding and decoding software that is available for virtually every programming environment and platform.
- XML is text-based and fairly easy to handle from low-tech programming environments.
- It's an extremely flexible format that can easily be extended in unambiguous ways.

To support extensibility, every element and attribute in XML has a namespace URI associated with it. This URI is specified using the xmlns attribute. Consider the following XML document:

```
<reverse_string
    xmlns="urn:schemas-develop-com:StringProcs"
>
  <string1>Hello, World</string1>
  <comment xmlns='http://foo.com/documentation'>
    This is a comment!!
  </comment>
</reverse_string>
```

The namespace URI for the <reverse_string> and <string1> elements is urn:schemas-develop-com:StringProcs. The namespace URI for the <comment> element is http://foo.com/documentation. The fact that the second URI is also a URL is immaterial. In both cases, the URI is simply used to disambiguate the <reverse_string>, <string1>, and <comment> elements from other elements that may accidentally share the same tag names.

XML allows namespace URIs to be mapped to locally unique prefixes as a convenience. This means that the following XML document is semantically equivalent to the previous one:

```
<sp:reverse_string
    xmlns:sp="urn:schemas-develop-com:StringProcs"
    xmlns:doc='http://foo.com/documentation'
>
  <sp:string1>Hello, World</sp:string1>
  <doc:comment>
    This is a comment!!
  </doc:comment>
</sp:reverse_string>
```

The latter form is considerably easier to author, especially if many namespace URIs are in use.

XML also supports typed data representation. The emerging XML Schema specification standardizes a vocabulary for describing XML data types. The following is an XML Schema description of the <reverse_string> element shown previously:

```
<schema
  xmlns='http://www.w3.org/1999/XMLSchema'
  targetNamespace='urn:schemas-develop-com:StringProcs'
>
  <element name='reverse_string'>
    <type>
      <element name='string1' type='string' />
      <any minOccurs='0' maxOccurs='*'/>
    </type>
  </element>
</schema>
```

This XML Schema definition states that the XML namespace urn:schemas-develop-com:StringProcs contains an element named <reverse_string> that contains a subelement named string1 (of type string), which is followed by zero or more unspecified elements.

The XML Schema specification also defines a set of built-in primitive data types as well as a mechanism for establishing the type of an element in an XML document. The following XML document uses the XML Schema type attribute to associate type names with elements:

```
<customer
   xmlns='http://customer.is.king.com'
   xmlns:xsd='http://www.w3.org/1999/XMLSchema'
>
  <name xsd:type='string'>Don Box</name>
  <age  xsd:type='float'>23.5</name>
</customer>
```

Additional mechanisms for linking XML document instances to XML Schema descriptions are being standardized at the time of this writing.

HTTP + XML = SOAP

SOAP codifies the use of XML as an encoding scheme for request and response parameters using HTTP as a transport. SOAP deals in a small number of abstractions. In particular, a SOAP method is simply an HTTP request and response that complies with the SOAP encoding rules. A SOAP endpoint is simply an HTTP-based URL that identifies a target for method invocation. Like CORBA/IIOP, SOAP does not require that a specific object be tied to a given endpoint. Rather, it is up to the implementor to decide how to map the object endpoint identifier onto a server-side object.

A SOAP request is an HTTP POST request. SOAP requests must use the text/xml content-type. Additionally, they must contain a Request-URI as per the HTTP specification. How the server interprets this Request-URI is implementation-specific, but many implementations are likely to use it to map to either a class or an object. A SOAP request must also indicate the method to be invoked using the SOAPMethodName HTTP header. The SOAPMethodName header is simply the application-specific method name scoped by a URI using a # character as a delimeter:

```
SOAPMethodName: urn:strings-com:IString#reverse
```

This header indicates that the method name is reverse and that the scoping URI is urn:strings-com:IString. The namespace URI that scopes the method name in SOAP is functionally equivalent to the interface ID that scopes a method name in DCOM or IIOP.

The HTTP payload of a SOAP request is simply an XML document that contains the values of the [in] and [in,out] parameters of the method. These values are encoded as child elements of a distinguished call element that shares the method name and namespace URI of the SOAPMethodName HTTP header. The call element must appear inside the standard SOAP <Envelope> and <Body> elements (more on these later). The following illustrates a minimal SOAP method request:

```
POST /string_server/Object17 HTTP/1.1
Host: 209.110.197.2
Content-Type: text/xml
Content-Length: 152
SOAPMethodName: urn:strings-com:IString#reverse
```

```
<Envelope>
 <Body>
  <m:reverse xmlns:m='urn:strings-com:IString'>
   <theString>Hello, World</theString>
  </m:reverse>
 </Body>
</Envelope>
```

The SOAPMethodName header must match the first child element under the <Body> element, otherwise the call must be rejected. This allows firewall administrators to reliably filter calls to a particular method without parsing the XML.

The SOAP response format is similar to that of the request. The response payload will contain the [out] and [in,out] parameters of the method encoded as child elements of a distinguished response element. This element's name is the same as the request's call element catenated with the Response suffix. The following is a minimal SOAP response to the request shown earlier:

```
200 OK
Content-Type: text/xml
Content-Length: 162

<Envelope>
 <Body>
  <m:reverseResponse xmlns:m='urn:strings-com:IString'>
   <result>dlroW ,olleH</result>
  </m:reverseResponse>
 </Body>
</Envelope>
```

In this case, the response element is named reverseResponse, which is simply the method name followed by the Response suffix. Also, note that the SOAPMethodName HTTP header is absent. This header is only required in the request message, not in the response.

Figure 6 The Other ORPC Request

Figure 6 and 7 show how SOAP maps onto the ORPC protocol concepts discussed earlier. What confuses many SOAP newbies is that there is no mandate for how a SOAP server will use the request header to dispatch the request; this is left as an implementation detail. Some SOAP servers will map Request-URIs to class names, dispatching the call to either static methods or to instances of the class that live for the duration of a request. Other SOAP servers will map Request-URIs to objects that are kept alive over time, often using the query string to encode a key that can be used to locate the object in the server process. Still other SOAP servers will use HTTP cookies to encode an object key that can be used to recover the state of an object at each method request. The key thing to remember is that the client is oblivious to these differences. The client software simply forms SOAP requests following the norms of HTTP and XML, leaving the server free to service the request in whatever manner it sees fit.

Figure 7 The Other ORPC Object Reference

Inside the SOAP Payload

The XML aspects of SOAP are simply an encoding scheme for serializing instances of data types into XML. To this end, SOAP does not mandate the use of a traditional RPC-style proxy. Rather, a SOAP method invocation consists of at least two data types: the request and the response. Consider this COM IDL fragment:

```
[ uuid(DEADF00D-BEAD-BEAD-BEAD-BAABAABAABAA) ]
interface IBank : IUnknown {
  HRESULT withdraw([in] long account,
                   [out] float *newBalance,
               [in, out] float *amount
           [out, retval] VARIANT_BOOL *overdrawn);
}
```

Under any RPC protocol, the values of the account and amount parameters would appear in the request message, and the values of the newBalance and overdrawn parameters would appear on the response, alongside the updated value of the amount parameter.

SOAP promotes the method request and method response to first class status. In SOAP, the request and response are actually instances of types. To understand how a method like IBank::withdraw maps to a SOAP request and response type, consider the following data type:

```
struct withdraw {
    long account;
    float amount;
};
```

This is simply a bundling of all of the request parameters into a single data type. Similarly, the following data represents the bundling of all of the response parameters into a single data type.

```
struct withdrawResponse {
    float newBalance;
    float amount;
    VARIANT_BOOL overdrawn;
};
```

Given the following simple Visual Basic program that uses the previously defined IBank interface

```
Dim bank as IBank
Dim amount as Single
Dim newBal as Single
Dim overdrawn as Boolean
amount = 100
Set bank = GetObject("soap:http://bofsoap.com/am")
overdrawn = bank.withdraw(3512, amount, newBal)
```

you can imagine that the underlying proxy (be it a SOAP, DCOM, or an IIOP proxy) would look something like

```
HRESULT Proxy_withdraw(long account,
           float *newBalance, float *amount,
           VARIANT_BOOL *overdrawn)
{
// serialize [in] params into a struct
    withdraw request;
    request.account = account;
    request.amount = *amount;
// send the request message
    proxy->Send(&request, sizeof(request));
// receive the response message
    withdrawResponse response = { 0 };
    proxy->Receive(&response, sizeof(response));
// deserialize response struct into [out] params
    *newBalance = response.newBalance;
    *amount = response.amount;
    *overdrawn = response.overdrawn;
    return S_OK; // or fail
}
```

Figure 8. Here, the parameters are serialized into a request object prior to sending the request message. Likewise, the parameters are then deserialized from the response object received in the response message. A similar transformation takes place on the server side of the call.

When invoking methods via SOAP, the request and response objects are serialized in a well-known format. Every SOAP payload is an XML document with a distinguished root element called <Envelope>. The tag name <Envelope> is scoped by the SOAP URI (urn:schemas-xmlsoap-org:soap.v1) as are all SOAP-specific elements and attributes. The SOAP envelope contains an optional <Header> element followed by a mandatory <Body> element. The <Body> element has one distinguished root element, which is either the request or the response object. The following is an encoding of an IBank::withdraw request:

```
<soap:Envelope
   xmlns:soap='urn:schemas-xmlsoap-org:soap.v1'>
  <soap:Body>
    <IBank:withdraw xmlns:IBank=
      'urn:uuid:DEADF00D-BEAD-BEAD-BEAD-BAABAABAABAA'>
      <account>3512</account>
      <amount>100</amount>
    </IBank:withdraw>
  </soap:Body>
</soap:Envelope>
```

The corresponding response message would be encoded as:

```
<soap:Envelope
   xmlns:soap='urn:schemas-xmlsoap-org:soap.v1'>
  <soap:Body>
    <IBank:withdrawResponse xmlns:IBank=
      'urn:uuid:DEADF00D-BEAD-BEAD-BEAD-BAABAABAABAA'>
      <newBalance>0</newBalance>
      <amount>5</amount>
```

```
      <overdrawn>true</overdrawn>
    </IBank:withdrawResponse>
  </soap:Body>
</soap:Envelope>
```

Notice that the [in, out] parameter appears in both messages.

After examining the format of the request and response objects, you may have noticed that the serialization format is generically:

```
<t:typename xmlns:t='namespaceuri'>
  <fieldname1>field1value</fieldname1>
  <fieldname2>field2value</fieldname2>
•••
</t:typename>
```

In the case of the request, the type is the implied C-style struct composed of the [in] and [in, out] parameters of the corresponding method. For the response, the type is the implied C-style struct composed of the [out] and [in, out] parameters of the corresponding method. This style of encoding using one child element per field is sometimes called element-normal form (ENF). In general, SOAP only uses XML attributes to convey out-of-band annotations that describe the information contained as element content.

Like DCOM and IIOP, SOAP supports protocol header extensions. SOAP uses the optional <Header> element to carry the information used by protocol extensions. Had the client-side SOAP software contained header information to send, the original request would have looked like Figure 9.

Figure 9 Using the SOAP Header

```
<soap:Envelope
   xmlns:soap=‘urn:schemas-xmlsoap-org:soap.v1’>
  <soap:Header>
    <causality xmlns="http://comstuff.com">
      <id>362099cc-aa46-bae2-5110-99aac9823bff</id>
    </causality>
  </soap:Header>
  <soap:Body>
    <IBank:withdraw xmlns:IBank=
      ‘urn:uuid:DEADF00D-BEAD-BEAD-BEAD-BAABAABAABAA’>
      <account>3512</account>
      <amount>100</amount>
    </IBank:withdraw>
  </soap:Body>
</soap:Envelope>
```

In this case, a header named causality was serialized with the request. Upon receiving the request, the server-side software can look at the namespace URI of the header and process the header extensions that it recognizes. Here, the header extension is identified by the http://comstuff.com URI and is expecting an object that looks like this:

```
struct causality {
  UUID id;
};
```

In the case of the request shown here, the header element can be safely ignored if its URI is not recognized.
You can’t safely ignore all SOAP payload headers. If a particular SOAP header is essential to the correct processing of the message, the particular header element can be marked as mandatory using the SOAP attribute mustUnderstand=’true’. This attribute informs the receiver that the header element must be recognized and processed to ensure proper functionality. To force the causality header shown earlier to be a mandatory header, the message would be written as follows:

```
<soap:Envelope
   xmlns:soap='urn:schemas-xmlsoap-org:soap.v1'>
  <soap:Header>
    <causality
          soap:mustUnderstand='true'
          xmlns="http://comstuff.com">
      <id>362099cc-aa46-bae2-5110-99aac9823bff</id>
    </causality>
  </soap:Header>
<!— soap:Body element elided for clarity —>
</soap:Envelope>
```

SOAP software that encounters an unrecognized mandatory header element must reject the message and indicate an error. If the server finds an unrecognized mandatory header element in a SOAP request, it must return a distinguished fault response and not dispatch the call to the target object. If the client finds an unrecognized mandatory header element in a SOAP request, it must return a runtime error to the caller. (In the case of COM, this would map to a distinguished HRESULT.)

Datatypes

Every element in a SOAP message is a SOAP structural element, a root element, an accessor, or an independent element. The soap:Envelope, soap:Body, and soap:Header are the only three structural elements in SOAP. Their basic relationship is described by the following XML Schema fragment:

```
<schema
 targetNamespace='urn:schemas-xmlsoap-org:soap.v1'>
  <element name='Envelope'>
    <type>
      <element name='Header' type='Header'
               minOccurs='0' />
```

```
      <element name='Body' type='Body'
               minOccurs='1' />
    </type>
  </element>
</schema>
```

Of the four types of SOAP elements, all but the structural elements are used to represent instances of a type, or references to instances of a type.

A root element is a distinguished element that is an immediate descendant of either the soap:Body or soap:Header element. soap: Body has exactly one root element, which represents the call, response, or fault object. This root element must be the first child element of soap:Body and its tag name and namespace URI must correspond to the HTTP SOAPMethodName header, or soap:Fault in the case of a fault message. The soap:Header element can have multiple root elements, one per header extension associated with the message. These root elements must be direct descendants of soap:Header and their tag name and namespace URI indicate the type of extension data that is present.

Accessor elements are used to represent fields, properties, or data members of a type. Each field of a given type will have exactly one accessor element in its SOAP representation. The tag name of the accessor corresponds to the field name of the type. Consider the following Java class definition:

```
package com.bofsoap.IBank;
public class adjustment {
  public int   account;
  public float amount;
}
```

Serialized instances of this class would look like the following within a SOAP message:

```
<t:adjustment
  xmlns:t='urn:develop-com:java:com.bofsoap.IBank'>
  <account>3514</account>
  <amount>100.0</amount>
</t:adjustment>
```

The accessors account and amount in this example are called simple accessors because they access values that correspond to primitive data types that are defined in Part 2 of the W3C XML Schema specification (see http://www.w3.org/TR/XMLSchema-2). This specification formalizes the names and representations of string, numeric, and date data types, as well as a mechanism for defining new primitive types using the <datatype> construct inside a new schema definition.

For accessors that refer to simple types, the value is simply encoded as character data directly below the accessor element as shown previously. For accessors that refer to compound types (those that are themselves structured using child accessors), there are two techniques for encoding the accessor. The simplest way is to embed the structured value directly below the accessor. Consider the following additional Java class definition:

```
package com.bofsoap.IBank;
public class transfer {
  public adjustment from;
  public adjustment to;
}
```

If the from and to accessors are encoded using embedded values, a serialized transfer object would look like this in SOAP:

```
<t:transfer
  xmlns:t='urn:develop-com:java:com.bofsoap.IBank'
>
  <from>
    <account>3514</account>
    <amount>-100.0</amount>
  </from>
  <to>
    <account>3518</account>
    <amount>100.0</amount>
  </to>
</t:transfer>
```

In this case, the values of the adjustment objects are encoded directly below their accessors.

There are several issues that need to be addressed when considering compound accessors. Consider the transfer class shown earlier. Both the from and to fields of the class are object references that potentially could be null. SOAP uses the XML Schemas null attribute to indicate null values or references. The following example shows a serialized transfer object whose from field is null:

```
<t:transfer
  xmlns:t='urn:develop-com:java:com.bofsoap.IBank'
  xmlns:xsd='http://www.w3.org/1999/XMLSchema/instance'
>
  <from xsd:null='true' />
  <to>
    <account>3518</account>
    <amount>100.0</amount>
  </to>
</t:transfer>
```

The implied value of the xsd:null attribute is false if it is absent. The nullability of a given element is controlled via the XML Schema definition. For example, the following XML Schema fragment would only allow the from accessor to be null:

```
<type name='transfer' >
  <element
    name='from'
    type='adjustment'
    nullable='true'
  />
  <element
    name='to'
    type='adjustment'
    nullable='false' <!— false is the default —>
  />
</type>
```

The absence of a nullable attribute in an element’s schema declaration implies that the element is not nullable in an XML document. The exact form of null accessors is currently being refined—consult the latest version of the SOAP specification for more information.

Another issue related to accessors is substitutability due to type relationships. Since the adjustment class shown previously is not a final class, it is possible that the from and to fields of the transfer object may actually refer to instances of derived types. To support this type-compatible substitution, SOAP uses the XML Schema convention of

a namespace-qualified type attribute. The value of this type attribute is a qualified name to the concrete type of the element. Consider the following class that extends the adjustment class:

```
package com.bofsoap.IBank;
public class auditedadjustment extends adjustment {
  public int   auditlevel;
}
```

Given the following Java-language fragment

```
transfer xfer = new transfer();
xfer.from = new auditedadjustment();
xfer.from.account = 3514; xfer.from.amount = -100;
xfer.from.auditlevel = 3;
xfer.to = new adjustment();
xfer.to.account = 3518; xfer.from.amount = 100;
```

the serialized form of the transfer object would look like the following in SOAP:

```
<t:transfer
  xmlns:xsd='http://www.w3.org/1999/XMLSchema'
  xmlns:t='urn:develop-com:java:com.bofsoap.IBank'
>
  <from xsd:type='t:auditedadjustment' >
    <account>3514</account>
    <amount>-100.0</amount>
    <auditlevel>3</auditlevel >
  </from>
  <to>
    <account>3518</account>
    <amount>100.0</amount>
  </to>
</t:transfer>
```

In this case, the xsd:type attribute refers to a namespace-qualified type name that the deserializer will use to instantiate the correct type of object. Because the to accessor referred to an instance of the expected type (instead of a substituted derived type), no xsd:type attribute is required.

The transfer class example just examined managed to sidestep one critical problem. What happens if the transfer object being serialized was originally initialized this way:

```
transfer xfer = new transfer();
xfer.from = new adjustment();
xfer.from.account = 3514; xfer.from.amount = -100;
xfer.to = xfer.from;
```

Based on the previous discussion, the serialized form of the transfer object would look like this in SOAP:

```
<t:transfer
  xmlns:t='urn:develop-com:java:com.bofsoap.IBank'>
  <from>
    <account>3514</account>
    <amount>-100.0</amount>
  </from>
  <to>
    <account>3514</account>
    <amount>-100.0</amount>
  </to>
</t:transfer>
```

This representation has two problems. The problem that is easiest to understand is that the same information is sent twice, resulting in a larger message size than is necessary. A subtler, but ultimately more important problem is that the identity relationship between the two accessors is lost since the deserializer cannot tell the difference between two adjustment objects with identical values and a single adjustment object referred to in two places. Had the receiver of this message performed the following test on the resultant object, the (xfer.to == xfer.from) test would never return true.

```
void processTransfer(transfer xfer) {
  if (xfer.to == xfer.from)
    handleDoubleAdjustment(xfer.to);
  else
    handleAdjustments(xfer.to, xfer.from);
}
```

The fact that (xfer.to.equals(xfer.from)) might return true only compares the values, not the identity of the two accessors.

To support serializing types that must maintain identity relationships, SOAP supports multireference accessors. The accessors I have examined so far are single-reference accessors; that is, the value is embedded below the accessor element and no other accessors are allowed to refer to the value. (This is similar to the concept of [unique] references in NDR.) Multireference accessors are always encoded as empty elements that contain only the well-known soap:href attribute. The soap:href attribute always contains a fragment identifier that corresponds to the instance that the accessor refers to. Had the to and from accessors been encoded as multi-reference accessors, the serialized transfer object would look like the following code:

```
<t:transfer
  xmlns:t='urn:develop-com:java:com.bofsoap.IBank'>
  <from soap:href='#id1' />
  <to   soap:href='#id1' />
</t:transfer>
```

This encoding assumes that an instance of a type that is compatible with the adjustment class has been serialized elsewhere in the envelope and that the instance has been tagged with the soap:id attribute as follows:

```
<t:adjustment soap:id='id1'
  xmlns:t='urn:develop-com:java:com.bofsoap.IBank'>
    <account>3514</account>
    <amount>-100.0</amount>
</t:adjustment>
```

For multireference accessors, it is the deserializer’s job to resolve the fragment identifiers (such as #id1) to the proper instance.

The previous discussion explained how a multireference accessor is associated with its target instance. What has yet to be explained is where the target instance is to be serialized. This is where the concept of an independent element and a package come into play.

Independent Elements

In SOAP, an independent element represents an instance of a type that is referred to by at least one multireference accessor. All independent elements are tagged by the soap:id attribute, and the value of this attribute must be unique throughout the SOAP envelope. Independent elements are encoded as if they were wrapped by an accessor whose tag name is the namespace-qualified type name of the instance. In the previous example, the qualified type name of the instance was t:adjustment.

SOAP restricts where independent elements can be encoded. SOAP defines an attribute (soap:Package) that can be applied to any element. This attribute is used to control where independent elements can be encoded. The SOAP serialization rules state that an independent element must be encoded as a direct descendant of either the soap:Header element, the soap:Body element, or any other element that's marked soap:Package='true'. By annotating an element as a package, you can guarantee that the XML element that encodes the instance is completely self-contained and has no multireference accessors to elements that are outside of the package.

Assume that the transfer class shown earlier corresponds to a method request. If the transfer type is not a package, the independent elements referred to by the to and from accessors would appear as direct descendants of the soap:Body element, as shown in Figure 10.

Figure 10 Direct Descendent of the SOAP Body

```
<soap:Envelope
  xmlns:soap='urn:schemas-xmlsoap-org:soap.v1'
  xmlns:t='urn:develop-com:java:com.bofsoap.IBank'>
  <soap:Body>
    <t:transfer> <!-- root elem of Body -->
      <from soap:href='#id1' />
      <to   soap:href='#id1' />
    </t:transfer>
    <t:adjustment soap:id='id1'> <!--ind. Elem-->
      <account>3514</account>
      <amount>-100.0</amount>
    </t:adjustment>
  </soap:Body>
</soap:Envelope>
```

Figure 11 Using the SOAP Package

```
<soap:Envelope
  xmlns:soap='urn:schemas-xmlsoap-org:soap.v1'
  xmlns:t='urn:develop-com:java:com.bofsoap.IBank'>
  <soap:Body>
    <t:transfer soap:Package='true'>
      <from soap:href='#id1' />
      <to   soap:href='#id1' />
      <t:adjustment soap:id='id1'> <!--ind. Elem-->
        <account>3514</account>
        <amount>-100.0</amount>
      </t:adjustment>
    </t:transfer>
  </soap:Body>
</soap:Envelope>
```

Had the transfer type been a legal SOAP package type, the encoding would have instead looked like the code in Figure 11. Notice that because the transfer element is a package, all of its multireference accessors refer to contained elements. This makes it easier to treat the transfer element as a distinct fragment of XML that can be separated from its parent.

There is one exception to the model in which multireference accessors always refer to independent elements. SOAP allows accessors containing string and binary data to be targets of multireference accessors. This means that the following is a legal SOAP fragment:

```
<t:mytype>
  <field1 soap:href="#id1" />
```

```
  <field2 soap:id="id1">Hello, SOAP</field2>
</t:mytype>
```

Despite the fact that the accessor2 element has a soap:id attribute, it is actually an accessor and not an independent element.

SOAP Arrays

Arrays are encoded as a special case of a compound type. An array in SOAP must have a rank (number of dimensions) and a capacity. An array is encoded as a compound type with each array element encoded as a subelement whose name is the namespace-qualified type name of the element.

Assume the following COM IDL type definition:

```
struct POINTLIST {
  long cElems;
  [size_is(cElems)] POINT points[];
};
```

An instance of this type would be serialized as follows:

```
<t:POINTLIST xmlns:t='uri for POINTLIST'>
  <cElems>3</cElems>
  <points xsd:type='t:POINT[3]' >
    <POINT><x>3</x><y>4</y></POINT>
    <POINT><x>7</x><y>5</y></POINT>
    <POINT><x>1</x><y>9</y></POINT>
  </points>
<t:POINTLIST>
```

Had the points field been marked with a [ptr] attribute, the encoding would use a multireference accessor and would look like this:

```
<t:POINTLIST xmlns:t='uri for POINTLIST'>
  <cElems>3</cElems>
  <points soap:href="#x9" />
</t:POINTLIST>
<t:ArrayOfPOINT soap:id='x9' xsd:type='t:POINT[3]'>
    <POINT><x>3</x><y>4</y></POINT>
    <POINT><x>7</x><y>5</y></POINT>
    <POINT><x>1</x><y>9</y></POINT>
</t:ArrayOfPOINT>
```

When encoding an array as an independent element, the tag name is the type name preceded by the ArrayOf prefix. Like NDR and CDR, SOAP supports partially transmitted arrays. If the number of child elements is less than the stated capacity, the elements are assumed to be missing from the end of the array. This can be overridden using the soap:offset attribute on the containing array element:

```
<t:ArrayOfPOINT soap:id='x9' xsd:type='t:POINT[5]'
                soap:offset='[1]'>
    <POINT><x>1</x><y>9</y></POINT>
</t:ArrayOfPOINT>
```

The soap:offset attribute indicates the index of the first element that appears in the array. In the previous example, elements 0 and 2 through 4 are not transmitted. SOAP also supports sparse arrays by annotating each element with its absolute index using the soap:position attribute.

```
<t:ArrayOfPOINT soap:id='x9' xsd:type='t:POINT[9]'>
<POINT soap:position='[3]'><x>3</x><y>4</y></POINT>
<POINT soap:position='[7]'><x>4</x><y>5</y></POINT>
</t:ArrayOfPOINT>
```

In this example, elements 0 through 2, 4 through 6, and 8 through 9 are not transmitted.
Please note that the precise syntax of arrays in SOAP is being re-examined at the time of this writing to adjust to the forthcoming W3C XML Schema specification. As always, consult the latest version of the SOAP specification for more details.

Faults

Occasionally, a server will not be able to properly service a method request. Sometimes this will be due to generic HTTP errors (say the Request-URI cannot be mapped to a local resource or there's an HTTP-level security violation). Sometimes this will be due to problems in the SOAP translation software such as marshaling errors or a mandatory header that cannot be recognized. Still other reasons are that a request cannot be properly serviced or the application/object code decides that it wants to return an application-level error to the caller. Each of these cases is explicitly dealt with in the SOAP specification.
If an error occurs at the HTTP level prior to dispatching the call to any SOAP code, a plain HTTP response must be returned. The standard HTTP status code numbering is used, with 400-level codes indicating a client-induced error, or a 500-level code indicating a server-induced error. This is typically handled automatically by the Web server software prior to your code executing.
Assuming that all is well at the HTTP layer, the next place where errors can occur is in the software that translates and dispatches the SOAP call to some application code (such as a COM object or CORBA servant). If an error occurs in this layer, the server must return a fault message in lieu of a standard response message. A fault message is simply an instance of the following type encoded as the root element of a soap:Body:

```
<schema
  targetNamespace='urn:schemas-xmlsoap-org:soap.v1'
>
  <element name='Fault'>
    <type>
      <element name='faultcode' type='string' />
      <element name='faultstring' type='string' />
      <element name='runcode' type='string' />
      <element name='detail' />
    </type>
  </element>
</schema>
```

The faultcode accessor must contain either a well-known SOAP fault code as an integer or a namespace-qualified value that is application-specific. The current SOAP fault codes are shown in Figure 12.

Figure 12 SOAP Fault Codes

Value	Name	Meaning
100	Version Mismatch	The call was using an unsupported SOAP version.
2	Must	An XML element was received that contained an

00	Understand	element tagged with mustUnderstand="true" that was not understood by the receiver.
300	Invalid Request	The receiving application did not process the request because it was incorrectly formed or not supported by the application.
400	Application Faulted	The receiving application faulted when processing the request. The detail element contains the application-specific fault.

The faultstring accessor contains the human-readable description of the error that occurred. The runcode accessor contains a string whose value must be Yes, No, or Maybe, indicating whether the requested operation was actually performed prior to the error generation. The detail accessor is optional, and is used to contain an application-specific exception object.
The following is an example of a SOAP fault message corresponding to a request containing an unrecognized mandatory header element:

```
<soap:Envelope
  xmlns:soap='urn:schemas-xmlsoap-org:soap.v1'
>
  <soap:Body>
    <soap:Fault>
      <faultcode>200</faultcode>
      <faultstring>
        Unrecognized 'causality' header
      </faultstring>
      <runcode>No</runcode>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Assuming that an application-specific fault needed to be returned, you might expect something more like the code that's shown in Figure 13.
Figure 13 Returning an Application-specific Fault

```
<soap:Envelope
  xmlns:soap='urn:schemas-xmlsoap-org:soap.v1'>
  <soap:Body>
    <soap:Fault>
      <faultcode>400</faultcode>
      <faultstring>
        Divide by zero occurred
      </faultstring>
      <runcode>Maybe</runcode>
      <detail>
        <t:DivideByZeroException xmlns:t="someURI">
           <expression>x = 2 / 0;<expression>
        </t:DivideByZeroException>
      </detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

In the case of an application-defined fault, the detail accessor plays the role of the soap:Body element for the application's exception/fault object.

Esoterica

One remaining HTTP-ism still needs to be addressed. SOAP supports (but does not require) the use of the HTTP Extension Framework conventions for specifying mandatory HTTP header extensions. These conventions serve two purposes. First, they allow an arbitrary URI to be used to scope a given HTTP header (as in XML namespaces). Second, these conventions allow mandatory headers to be distinguished from optional headers (as in soap:mustUnderstand). The following is an example that uses the HTTP Extension Framework to distinguish the SOAPMethodName header as a mandatory header extension:

```
M-POST /foobar HTTP/1.1
Host: 209.110.197.2
Man: "urn:schemas-xmlsoap-org:soap.v1; ns=42"
42-SOAPMethodName: urn:bobnsid:IFoo#DoIt
```

The Man header maps the SOAP URI to the header prefix 42 and indicates that servers that do not recognize SOAP must return an HTTP error with a status code of 501 (Not Implemented) or 510 (Not Extended). The HTTP method must be M-POST, indicating that mandatory header extensions are present.

Conclusion

SOAP is a typed serialization format that happens to use HTTP as a request/response messaging transport. SOAP was designed to work well with the emerging XML Schema specification, and supports interoperation between COM, CORBA, Perl, Tcl, the Java-language, C, Python, or PHP programs running anywhere on the Internet.

I hope that I've given you a clearer understanding of the specifics of the protocol. I encourage you to experiment with SOAP either by trying one of the SOAP-enabled systems listed at http://www.develop.com/soap/ or by hacking something up yourself. I found that it takes me less than an hour to get a basic SOAP client and server up and running using my scripting language of choice (JScript). Your mileage may vary depending on your familiarity with HTTP and XML and the maturity of your target platform.

For related articles see:
http://msdn.microsoft.com/xml/general/soapspec-v1.asp
http://www.microsoft.com/mind/0100/soap/soap.asp
The author recommends:
http://www.w3.org/XML
http://www.develop.com/soap
Background information:
The author recommends:
IIOP Complete, William Ruh, Thomas Herron, Paul Klinker (Addison Wesley);
Computer Networks, Andrew Tannenbaum (Prentice Hall)

Don Box is a cofounder of DevelopMentor, a COM think tank that educates the software industry in COM, MTS, and ATL. Don wrote *Essential COM*, and coauthored the follow-up *Effective COM* (Addison-Wesley, 1998). Reach Don at http://www.develop.com/dbox.

From the March 2000 issue of MSDN Magazine.
Get it at your local newsstand, or better yet, *subscribe*.