Understanding SOAP

A standard for packaging information in XML, SOAP eases the development of distributed applications.

By Kenn Scribner

PC Online
MORE ON THE WEB
www.pcmag.com
▶ For more solutions for developers and designers, please visit us online at *www.pcmag.com/internetpro*.

Distributed computing is a hot topic today. To facilitate business-to-business and business-to-consumer interaction, the ability for disparate applications and systems to communicate is more important than ever. But previous solutions for distributed computing were not created for heterogeneous environments like the Internet. A new approach was needed; thus the Simple Object Access Protocol, SOAP, was designed as "a lightweight protocol for information exchange in a decentralized, distributed environment." (For more information, go to *www.w3.org/tr/soap/*.)

The protocol defines a messaging mechanism for encoding information into an XML wrapper. Quite often, SOAP is used in a more specific way—to interpret a remote method's parameter values at runtime and stuff those values into an XML document using a specified layout. The XML data is then transported to the remote server using HTTP, although other transport protocols may also be used. These days, we give that remote method a catchy name, calling it a *Web service*. Of course, there are many other specifications for remote method invocation (CORBA's IIOP, DCOM's ORPC, and Java Remote Method Protocol, for example). SOAP's particular—and huge—benefit is that it's text-based (through XML) rather than binary, and not specific to any vendor.

With SOAP, distributed computing is simply a matter of consuming resources on a remote computer as if the remote computer and the calling (local) computer were the same machine. The goal is to tie the distributed systems together seamlessly, so that when you call a given method, you don't know (and presumably don't care) whether the call is actually handled by a remote system. In real-world situations, you very often do care, if only because the call latency is greatly increased for calls executed remotely; the method simply takes longer to complete its task. But for now, let's ignore latency issues and imagine that SOAP does seamlessly integrate distributed systems.

The fact that SOAP uses a common transport protocol—almost always HTTP—is SOAP's secret weapon and one of the sources of its power. Because blocking of HTTP data by corporate firewalls is almost unheard of, SOAP (as bound to HTTP) should easily pass through firewalls. Proprietary distributed-computing protocols can't make that claim. The network addresses they use are usually blocked to secure systems against entry by unscrupulous hackers. The other source of

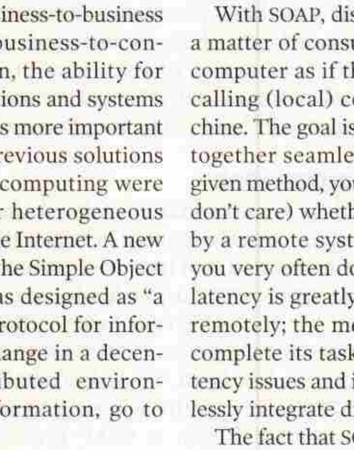


FIGURE 1: **SOAP GENERALLY** uses HTTP as its transport protocol, and as a result it's not stopped at corporate firewalls.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap
.org/soap/envelope/" xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance" xmlns:xsd="http://www.w3
.org/2001/XMLSchema">
  <soap:Header>
<AccountNum xmlns="http://www.mycompanyurl.com/">
11285-990</AccountNum>
  </soap:Header>
  <soap:Body xmlns="http://www.mycompanyurl.com/">
    <IssuePurchaseOrder>
      <po>PO information here...</po>
    </IssuePurchaseOrder>
  </soap:Body>
</soap:Envelope>
```

FIGURE 2: **A REPRESENTATIVE** SOAP request packet.

SOAP's power is XML. SOAP provides the ability to integrate systems that previously didn't communicate and share resources. How? As it happens, XML is quickly becoming understood by computing systems nearly everywhere. If you can find a way to get an XML document into a system, odds are there is software available to enable the system to read and interpret the XML-encoded information. For remote-computing purposes, SOAP was designed to take a method's parameters in their native binary form and carry those parameters as XML information to the remote server, where a corresponding SOAP processor pulls the XML information and return it to its binary state for processing.

The SOAP Envelope serves as a wrapper for the really important information.

A CLOSER LOOK

So let's see what makes SOAP tick. In Figure 1 you can see the relationship between the SOAP-formatted XML and its carrier protocol, in this case HTTP. HTTP uses a header to denote the host system to which this packet is directed, the content of the packet (XML-based text in this case), and so forth. There is a SOAP-specific HTTP header entry to facilitate automated packet routing within a SOAP processing architecture. Other protocols have similar requirements.

The SOAP Envelope serves as a wrapper for the really important information. Note that unlike a postal envelope, the SOAP Envelope doesn't specify an address. Think of the SOAP Envelope as a three-ring binder with groups of notes and information separated by tab sheets. The tabbed sections are an optional message Header element, the message Body element, and any custom XML elements you might insert.

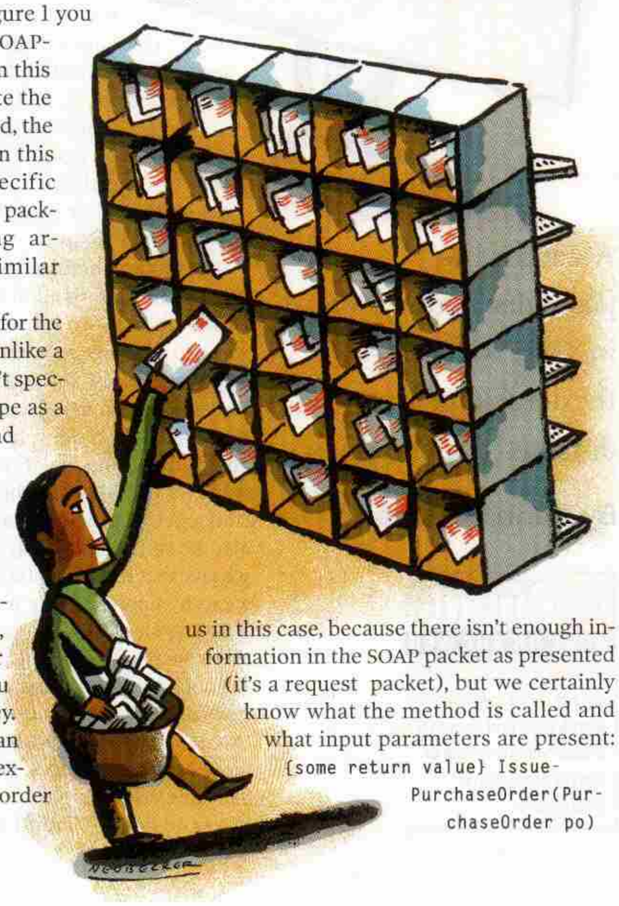
The SOAP Header, if there is one, contains management and control information, such as an account number or customer identifier. The SOAP Body is where you record the information you wish to convey. To make this a bit more concrete, here's an outline of a typical SOAP message. In this example, imagine we're sending a purchase order to a remote method for processing.

```
- Envelope
  - Header
    - Account number
  - Body
    - Issue a purchase order
      - PO information
```

The simplified outline isn't terribly different from the actual SOAP XML, shown in Figure 2. Here you see the Envelope node that contains the SOAP Header and Body. The Header node is designed to contain meta-information such as the algorithm used to encrypt the body, or perhaps the public key necessary to unlock the encrypted text. Or the header might contain a transaction identifier, a causality identifier (a deadlock prevention mechanism), or any other out-of-band information. In this case, the header contains an account number. The Body contains a child node, named `IssuePurchaseOrder`, which happens to be the name of the remote method.

The SOAP specification requires the first child of the Body element to be the method in question. Remember that SOAP is intended to be—or was initially designed, at least, to be—a remote procedure invocation protocol. That being the case, the underlying assumption is that a method on a remote computer is to be executed on your behalf. We may not know what the method will return to us in this case, because there isn't enough information in the SOAP packet as presented (it's a request packet), but we certainly know what the method is called and what input parameters are present:

```
{some return value} Issue-
        PurchaseOrder(Pur-
           chaseOrder po)
```



We know this because the SOAP specification tells us that the first child of the body has an XML node name that is the same as the method name. Any children of the method node are similarly named for the method parameters. Since XML also has the ability to represent data types, such as strings and integers, we can process the parameter data appropriately.

Based on the method name in our example, you can probably guess that the SOAP method will accept an input purchase order and return some form of processing confirmation. That being the case, the remote computer might return a SOAP packet much like this outline:

```
- Envelope
  - Body
    - IssuePurchaseOrderResponse
      - IssuePurchaseOrderResult
        - Results here...
```

Of course, the response is also encoded in XML, as in Figure 3. Here you see the same SOAP Envelope and Body nodes, but the `IssuePurchaseOrder` method node names have changed. The method return names themselves are shown as they're conventionally generated. It's the position in the XML hierarchy that matters here. Once again, the method's return information is encoded beginning as the first child of the SOAP Body node. That node's first child must be the return value from the method. Since the return value has no name—it's simply returned from the method invocation—by convention it's named *result* or some derivative, such as the method name concatenated with *Result*, as you see in Figure 3. Here, the method returned a Boolean value. If the method had output parameter values, you would find those encoded as sibling nodes of the result node.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap
.org/soap/envelope/" xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance" xmlns:xsd="http://www.w3
.org/2001/XMLSchema">
  <soap:Body xmlns="http://www.mycompanyurl.com/">
    <IssuePurchaseOrderResponse>
      <IssuePurchaseOrderResult>true
      </IssuePurchaseOrderResult>
    </IssuePurchaseOrderResponse>
  </soap:Body>
</Envelope>
```

FIGURE 3: **A REPRESENTATIVE** SOAP response packet.

When combined with WSDL, SOAP becomes even more powerful and flexible.

A DIFFERENT TWIST

The way we've described SOAP is based on the SOAP specification, and the fact we've described request and response packets indicates that we're referring specifically to SOAP in the remote-procedure-call sense—the way SOAP began. Back then, there was no way to send a remote site anything that deviated from the SOAP specification. You encoded the data according to the specification, because if you didn't, the receiving side probably wouldn't be able decipher the content.

Today, though, we use the SOAP protocol in a broader sense. In addition to *RPC SOAP*, we also use *Messaging SOAP*. *RPC SOAP* generally refers to SOAP encoded according to the SOAP specification, sections 5 and 7. *Messaging SOAP*, however, is more freeform. Information is encoded in nearly any fashion, as dictated by the processing server. But if the information is encoded any way we like, or nearly so, how do we communicate the syntax of the packet to potential consumers?

The answer to that lies with a protocol even younger than SOAP itself, the Web Services Description Language (WSDL). WSDL identifies the constituent pieces of a SOAP packet and describes how they're arranged within the packet. And because WSDL is also XML-based, we can process this information algorithmically, just as with SOAP itself.

WSDL describes three other use models for the SOAP protocol besides the common request/response scenario. With request/response, a client requests some action, and the server responds with either a successful result or an error or exception. There are two endpoints at work here—the client and the server.

The other models are *one-way SOAP*, where the client sends information to a server without regard to a response (like an e-mail note), *solicit-response*, where the server requests information from the client (as in a status check), and *notification*, where the server sends information (like an event) to the client without regard to a response. Of course, which model you're using depends upon your point of view. At any given time you may find yourself both a client and a server.

When combined with WSDL, SOAP becomes even more powerful and flexible. In fact, this powerhouse combination is the basis for Web services. As with any technology, the details associated with SOAP are many and varied, but the truth is that SOAP is simple and easy to use. If you have a Web Server, along with access to a network and client computers, and can process XML, you can create a SOAP server in a relatively short time, at least for a single SOAP method. Today, though, most people turn to fully functional systems designed to process SOAP-based information. There are turnkey solutions available for IBM mainframes, Linux PCs, Windows machines, and a host of other systems. If such a solution meets your requirements, you can save development time by using it. If not, at least working with SOAP isn't terribly difficult, and most people seem to find it fun and interesting.

Kenn Scribner is a Wintellect consultant and instructor specializing in SOAP, XML, and Microsoft .NET XML Web services. He has written or contributed to a number of programming books.