# 14
# Making the Right Choices for SOAP Scalability

Software developers live in a time that offers the greatest choice of software development tools, application servers, and connectivity ever. Each choice that you make affects the scalability and reliability of your finished application, especially if you're building Web services. For example, as you will learn later in this chapter, my study of SOAP encoding styles found a 30-fold performance improvement by choosing one SOAP encoding style over the others. By understanding the performance impact of SOAP encoding styles, Web service development tools, application servers, and platforms, our choices greatly improve system performance.

This chapter presents results of an investigation that shows how each choice immediately impacts scalability and reliability. It discusses the impact of letting development tools make choices for us on scalability and performance. Then it presents directions, tools, and test agents to stage tests in your own environment.

## Why is SOAP So Popular?

In my experience, when independent technology innovations intersect, the world enjoys life-changing products, services, and techniques. For example, the light bulb required both electricity generation and filament fiber technology. In the case of Web services, enterprise information technology managers had just come off a multiyear binge where they bought huge numbers of

computers, servers, routers, and other Web infrastructure. Left with a recession, terrible stock market, uncertainty about the world economy, and having to contend with terrorism and SARS, these information managers returned to their existing Web infrastructures to increase productivity of their teams through new software projects.

At the same time, most software developers realized they really liked XML. For example, XML was much better than using the Microsoft Windows Registry or text-based property files to store and describe application data. Software developers saw a good thing in XML and wanted to find more ways to use XML in their applications.

As a software developer I began noticing APIs that expected to receive a value that contained XML encoded data. For example, when building a portal system for Sun Microsystems, I found that the servlet to create a new user account received the fields that made up the user contact information (email, address, and telephone number) in an XML document. Rather than pass in one value at a time to a method, instead the method took one XML value that contained several values. Using XML to implement an application's interfaces is a clear win to developers. Plus, these XML-described interfaces could work across platforms and programming languages. With XML everything looks like an interface.

These intersecting technologies power the widespread enthusiasm for Web services. At the same time, software developers were again experimenting with software architectures, especially with the location of application business logic and presentation code. Presentation code handles windows, mice, keyboard, and other user interactions. Business logic is the instructions that define the behavior and operation of an application.

The first-generation software architecture built the presentation and business logic on a single system. In the second generation, client/server architecture brought back the large, centrally controlled datacenter so familiar in the 1960s when mainframes ruled the information world. In client/server architecture, the desktop system is a "dumb" terminal that only needs to display the data provided by the server. The early Internet was modeled after client/server architecture where the browser made simple requests to a server. As browser's improved in functionality—applets, JavaScript, ActiveX, and DHTML were introduced—some systems included business logic on the desktop side. However, the majority of function remained on the server.

The age of “grid computing” is upon us where an application hosts business logic modules on the desktop or server. The modules discover each other using UDDI and P2P technologies. Also multiple copies of the business logic modules may run in a grid of datacenters to allow failover, dynamic routing, and functional specialization. All of these architectures run in a Web environment and can host Web services.

So even if SOAP-based Web services are replaced with some other type of Web service technology, remote procedure calls using XML encoded data will be around for a very long time.

## SOAP Encoding Styles

SOAP uses XML to marshal data that is transported to a software application. Most of the time, SOAP moves data between software objects, but the SOAP specification was intended to be useful for old legacy systems as well as modern object-oriented systems. Consequently, SOAP defines more than one data encoding method to convert data from a software program into XML format and back again. The SOAP encoded data is packaged into the body of a message and sent to a host. The host then decodes the XML-formatted data back into a software object.

Since SOAP’s introduction, three SOAP encoding styles have become popular and are reliably implemented across software vendors and technology providers:

- *SOAP RPC encoding,* also known as Section 5 encoding, as defined by the SOAP 1.1 specification and later defined in SOAP 1.2 as RPC encodings and conventions.
- *SOAP Remote Procedure Call Literal encoding (SOAP RPC-literal)* uses RPC methods to make the call but uses an XML do-it-yourself method for marshaling the data.
- *SOAP document-style encoding,* also known as message-style or document-literal encoding.

There are other encoding styles, but software developers have not widely adopted them, mostly because their promoters disagree on a standard. For example, early on in the invention of Web services, Microsoft promoted Direct Internet Message Exchange (DIME) to encode binary file data, while

the rest of the computer industry adopted SOAP with Attachments. SOAP RPC encoding, RPC-literal, and document-style SOAP encoding have emerged as the encoding styles that a software developer can count on.

Some developers do not realize that such encoding styles exist because the tools they use to develop Web services are doing the work of implementing the encoding styles for the developer. For example, BEA WebLogic Workshop provides a fast and efficient implementation of the JWS interface. JWS implements a set of APIs and a standard description of the files the JWS engine needs to automatically deploy the Web service on the server. JWS builds the Web service deployment descriptors for you automatically. So you need to only define the public Java methods and JWS publishes the SOAP proxy to access the methods. This makes development appear very easy, but there is a lot of work going on under the covers, so to speak. We'll see this in more depth later in this chapter.

Before I discuss SOAP encoding style's impact on performance, you should understand the differences between these styles of SOAP encoding. Figure 14–1 shows the entire stack for a SOAP RPC encoded call.

SOAP RPC is the encoding style that offers the most simplicity for developers. The developer makes a call to a remote object, passing along any necessary parameters. The SOAP stack serializes the parameters into XML, moves the data to the destination using transports such as HTTP and SMTP, receives the response, deserializes the response back into objects, and returns the results to the calling method. Whew! SOAP RPC handles all the encoding and decoding, even for very complex data types, and binds to the remote object automatically.

Now, imagine you are a developer with some data already in XML format. SOAP RPC also allows literal encoding of the XML data as a single field that is serialized and sent to the Web service host. Since there is only a single parameter—the XML tree—the SOAP stack only needs to serialize one value. The SOAP stack still deals with the transport issues to get the request to the remote object. The stack binds the request to the remote object and handles the response.

Lastly, in a SOAP document-style call, the SOAP stack sends an entire XML document to a server without even requiring a return value. The message can contain any sort of XML data that is appropriate to the remote service. In SOAP document-style encoding, the developer handles everything,

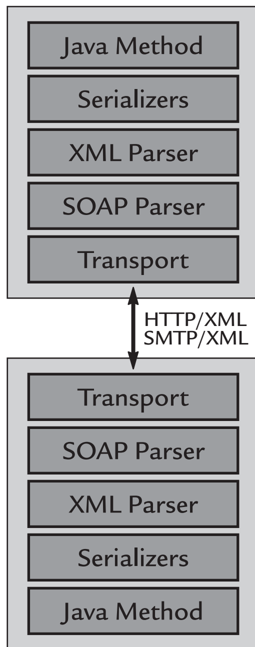


**Figure 14–1** A Java method calls a Web service by using the SOAP stack and SOAP RPC encoding.

including determining the transport (e.g., HTTP, MQ, SMTP), marshaling and unmarshaling the body of the SOAP envelope, and parsing the XML in the request and response to find the needed data.

The three encoding systems are compared in Figure 14–2.

SOAP RPC encoding is easiest for the software developer; however, all that ease comes with a scalability and performance penalty. SOAP RPC-literal encoding is more involved for the software developer to handle XML parsing, but requires fewer overheads from the SOAP stack. SOAP document-literal encoding is most difficult for the software developer, but consequently requires little SOAP overhead.

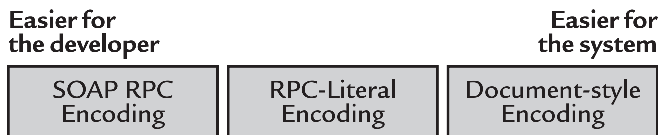


**Figure 14–2** SOAP encoding styles offer software developers greater productivity, but it comes at a performance and system resource cost.

Why is SOAP RPC easier for the developer? With this encoding style, you only need to define the public object method in your code once; the SOAP stack unmarshals the request parameters into objects and passes them directly into the method call of your object. Otherwise, you are stuck with the task of parsing through the XML tree to find the data elements you need, and then you get to make the call to the public method.

There is an argument for parsing the XML data yourself: Since you know the data in the XML tree best, your code will parse that data more efficiently than generalized SOAP stack code. As we will see when we measure scalability and performance in SOAP encoding styles, we will find this to be the case.

But before I go further into that, we look at how enterprise information systems managers are coming to grips with SOAP encoding styles and scalability.

## Simple Object Access Needs Simple Testing

Elsevier (www.elsevier.com) is the leading research content publisher for the science, technology, and medical industries. Elsevier now uses a content publishing platform that uses SOAP to build application programming interfaces. Elsevier's information managers need to know if their choices of SOAP encoding style will scale and perform to handle millions of transactions every day. Their decisions affect how Elsevier will invest capital in new infrastructure. Over time, they need to know how new releases of their own software, new releases of application server software, and platform changes will affect scalability and performance.

Elsevier learned about TestMaker through the open source community and contacted PushToTest (details at www.pushtotest.com) to see if TestMaker was appropriate for their testing needs. Elsevier asked PushToTest to conduct an independent audit of SOAP stacks and encoding styles to answer their questions about system performance and scalability. The resulting test environment is illustrated in Figure 14–3. PushToTest delivered a Test Web Service (TWS) that handles RPC, RPC-literal, and document-style SOAP messages and runs on a variety of application servers. The environment is completed with a set of intelligent test agents to check TWS for scalability and performance.

TestMaker checks Web services for scalability, performance, and reliability. Software developers, QA analysts, and IT managers use TestMaker to

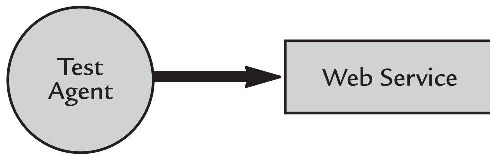


**Figure 14–3** The test environment to check SOAP scalability uses test agents to make requests to a SOAP-based Web service.

build intelligent test agents that implement archetypal user behavior. The agents drive a Web service using native protocols (HTTP, HTTPS, SOAP, XML-RPC, SMTP, POP3, IMAP) just as a real user would. Running multiple intelligent test agents concurrently creates near production-level loads to check the system for scalability and performance.

In addition to checking SOAP encoding scalability, the Elsevier test environment provides a benchmark specific to Elsevier's systems to show a performance comparison for a variety of application servers and platforms. For example, TWS is currently implemented to run on IBM WebSphere, BEA WebLogic Workshop, and the SunONE application server. I am confident that ports to webMethods Glue, Apache Axis, Systinet WASP, and other application servers is straightforward.

I built the Elsevier test environment by customizing TestMaker to support SOAP RPC, SOAP RPC-literal, and SOAP document-style requests, and by implementing TWS to respond to requests in these encoding styles. The request to TWS contains two parameters: the first defines the size of the response and the second defines a `delay` value before responding. TWS responds by creating a response document containing random gibberish words that appeared in five response elements; each element has one child element. A TestMaker test agent uses the Apache SOAP library to make requests to TWS. The test agent varied the number of concurrent requests to TWS and the payload size of the response. The test agent logged the results to a delimited log file, which was subsequently summarized by a tally script. The tally script determined the number of TPS performed by the test by counting the duration of successful transactions. We defined success as the absence of transport or SOAP faults.

With Sun Microsystems support, I ran the tests on Sun Solaris E4500 servers with six CPUs and 4 GB of RAM. The TWS used the SOAP stack provided by the underlying application server. For example, WebSphere provides Apache SOAP, BEA WebLogic provides their own implementation

that uses the JAX-RPC APIs, and the SunONE application server uses the Java 1.4 JAX-RPC library. On the client side TestMaker uses the Apache SOAP library.

In the Elsevier project, I found that a developer's choice of encoding style determines to a large extent the scalability and performance of a Web service. The SOAP implementations universally showed scalability problems when using SOAP RPC encoding, especially as payload sizes increased, as illustrated in Figure 14–4.

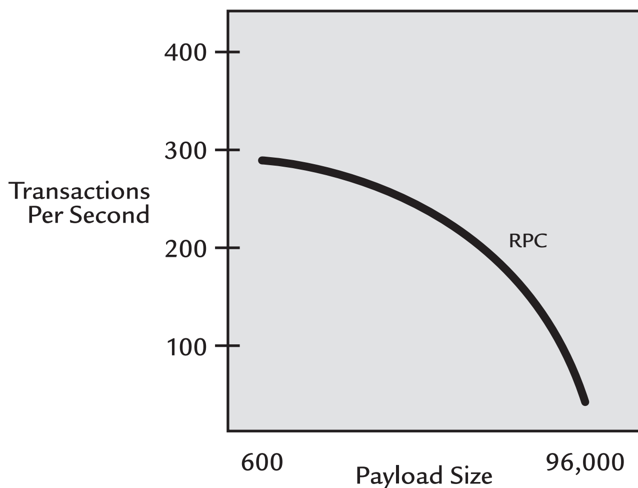


**Figure 14–4** SOAP RPC encoding: Scalability problems become noticeable with increased payload sizes.

As Figure 14–4 shows, the test agent recorded 294 transactions per second when making requests where the response SOAP envelope measured 600 bytes of SOAP RPC-encoded data. As the test agent increased the response size, the transactions per second plummeted. When making requests of 96,000 bytes of SOAP RPC-encoded data, the agent measured only 9.5 transactions per second.

When the test environment uses SOAP document-style encoding, the performance fared much better, as you can see in Figure 14–5.

With 600 bytes of document-encoded data, the test agent measured 469 TPS. Recall that the SOAP RPC-encoded requests gave us 294 TPS for requests of the same size. Additionally, when the test agent increased the

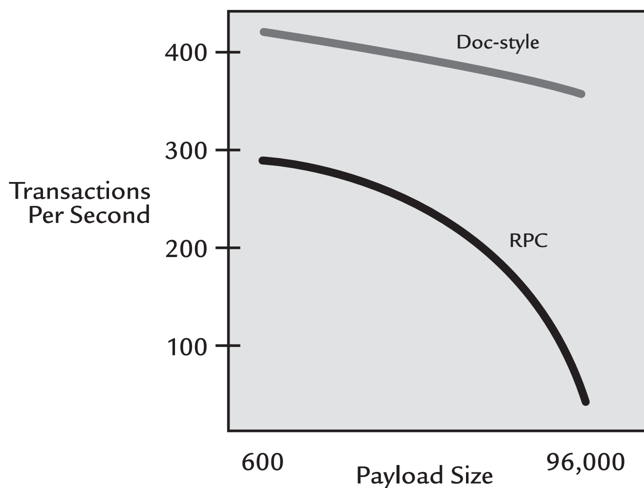


**Figure 14–5** Document-style encoding: Performance stays relatively stable with increased payload sizes.

response size, the TPS values did not degrade significantly when we used document-style encoded responses.

When the test environment uses SOAP RPC-literal encoding, I found an efficient middle ground, as you can see in Figure 14–6.

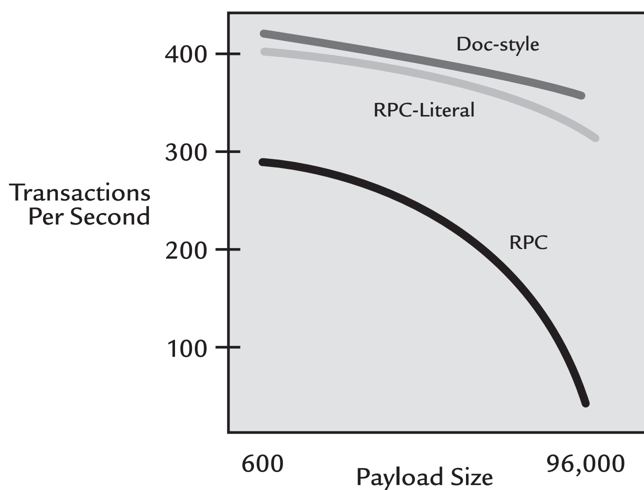


**Figure 14–6** SOAP RPC-literal provides the performance benefits of SOAP document-style encoding with a little more work required to parse through the XML data.

## Should You Let the Tools Do the Driving?

While the SOAP encoding styles provide a good range of power and flexibility, they also introduce interoperability problems. Most of the SOAP tools on the Java platform default to SOAP RPC encoding styles. For example, when using IBM WebSphere Application Developer, the default encoding style is set to SOAP RPC. On the other side of the divide, .NET development tools implement document-style SOAP calls by default. This is akin to watching two boats pass in the night. Both can be made to interoperate, but developers need to be wise to the different encoding styles to avoid problems.

In their attempt to make software developers' lives easier, these tools may be making decisions for you that affect scalability. This was highlighted when Microsoft and Sun debated the relative virtues of J2EE and .NET at an event hosted in Silicon Valley by the Software Development Forum. Details are found at http://www.sdforum.org. Microsoft made the argument that it serves developers best by being the sole supplier of a complete solution. On the other end of the spectrum, Sun posited that developers should have a choice of tools that they can assemble into a solution. This top-down versus bottom-up argument permeates into both companies' development tools. For example, representatives from Sun and Microsoft were asked to explain why developers would choose SOAP RPC encoding over SOAP document-style encoding. Microsoft's reps gave a somewhat technical answer, but conceded that they thought the issue was moot, since developers should rely on their development tools to make decisions of encoding styles.

Software developers serve themselves best by making informed decisions of how helpful their development tools and environments should be. Understanding each tool's handling of SOAP encoding styles is an important factor to delivering well performing and reliable software projects.

### May I Freak Out Now?

With 600 bytes of SOAP RPC-literal encoded data, the test agent measured 422 TPS. That is nearly the performance recorded for SOAP document-style requests. SOAP RPC-literal encoding did not show the plummeting TPS function of SOAP RPC encoding performance.

At this point your mind might be racing with questions like:

- Is this true? Can SOAP scalability be that bad?

- What are the performance differences between the application servers? Will IBM WebSphere perform better than BEA WebLogic?
- What are the most important considerations to build reliable Web services?
- What equipment do I need?
- What is the best way to deploy Web services?
- What is the best software platform to build my Web service?

In my experience every production environment is unique. So, rather than try to be your answer guy for every application server and encoding style, I would like to give you a performance kit that you can download and use in your own production environment. I have made a generalized version of the Elsevier test environment available for free download for your immediate use at http://www.pushtotest.com/ptt/kits/encodingkit.html. In the next section I present the kit and show how you may use it in your own environment.

## The Performance Kit

Enterprise information manager choices for Web service infrastructure are critical to deploying well performing, highly reliable, and scalable systems for running Web services. I receive calls everyday from software developers, QA technicians, and IT managers wanting to know:

- What are the most important considerations to build reliable Web services?
- What equipment do I need?
- What is the best way to deploy Web services?
- What is the best software platform to build my Web service?

Several things prevent me from having an immediate answer to your questions. These include the following:

- The application servers you use are a moving target. The popular commercial and open source software vendors work tirelessly to improve performance, add functions, and improve techniques to use their products and technology. The

performance and scalability characteristics change with each new version of their products.
- The underlying code libraries in a SOAP stack are moving targets too. For example, most of the Java-based Web service applications use the popular Apache Xerces library to parse XML data. Xerces is both complex and slow. Changing to a newer version of Xerces or replacing it with another library greatly impacts Web service performance and scalability.
- Your network infrastructure (load balancers, routers, server equipment, storage devices) is unique from everyone else. This makes an apples-to-apples comparison very difficult.
- Your own software development efforts will improve performance and scalability as new builds of your software application are applied to your production environment.

So rather than try to answer your questions, this section shows how to understand Web service scalability and performance in your own environment.

The Web Services Performance Kit helps information managers with reliable and repeatable performance measurement tools for systems that drive Web service-based business. The kit may be applied to any Web service-based system. However, the examples are specific to the SunONE application server, BEA WebLogic server, and IBM WebSphere application server. First I give an overview of the kit's contents and then we will see how the kit is applied to an application server environment.

## How Do I Get the Performance Kit?

The Web Services Performance Kit is available for free download at the PushToTest Web site. Point your favorite browser to http://www.pushtotest.com/ptt/kits/index.html to download the kit. The PushToTest site features the latest software, archives of previous versions, frequently asked questions about the kit, and various technical support services, including email support lists for users and developers.

The kit is distributed in a Zip archive file format. Extracting the file contents installs everything in the kit. You will need to also download TestMaker from the PushToTest site. TestMaker is fully described in Chapter 5. The kit contains test agents that run in the TestMaker environment and the Test Web Service that runs everywhere Java 1.4 or greater runs, including Windows,

Linux, Solaris, and Macintosh OS X. Other Java-enabled platforms are capable of running the kit but have not been tested.

TestMaker updates appear generally once a month to offer bug fixes and new features. The PushToTest Web site offers a free email announcement service to send email alert messages when new versions become available. (PushToTest does not publish or share email addresses.)

### Installing the Performance Kit

Installing the kit is very simple. Balancing, that is, the Test Web Service, which is often very difficult to install and configure. Inside the Performance Kit you will find the following things.

- Easy-to-read instructions describing how to build and test Web services on the SunONE application server, BEA WebLogic server, and IBM WebSphere server. In this chapter, due to space considerations, I limit the examples to testing on a BEA WebLogic Workshop.
- TWS—a free, open source Web service that simulates a database-driven Web service. TWS responds to a variety of SOAP encoding styles and on several application servers.
- Performance_Agent—a free intelligent test agent that will determine the scalability and performance of the TWS. Performance_Agent makes multiple concurrent requests to TWS and measures performance in a transactions-per-second result.
- TestMaker—a free, open source utility to run Performance_Agent and to create agents for your own Web service testing. See Chapter 5 for details on TestMaker.

The Performance Kit comes as a compressed archive in ZIP format of HTML documents, test agent script files, and Java source code files. Please use a decompression utility to extract the files onto your local computer. The index.html document, among the decompressed files, contains installation, configuration, and "how-to" instructions for the kit.

PushToTest provides a variety of support options to help you use the kit. Support is offered for free from the PushToTest community. Additionally,

for-pay support agreements are offered. Details on both are at http://www.pushtotest.com/ptt/support.html.

## Getting Started

After you unpack the kit, I recommend you spend a few minutes getting to know its components. Take a look at the documentation and peruse the source code files. Most of the code is self-documented. Then follow these steps to run the kit:

1. If you have not already done so, download and install one of the supported application servers. Sun, BEA, and IBM feature free trial versions of their software: SunONE application server, BEA WebLogic server, and IBM WebSphere application server. This chapter looks specifically at BEA WebLogic Workshop.
2. Install the Test Web Service.
3. Download and install TestMaker.
4. Run the PerfCheck agent to test TWS for performance. PerfCheck is an intelligent test agent written and run with TestMaker. PerfCheck includes a Tally script that shows the TPS and other performance metrics for the TWS.

Next we look at the components of the kit and see what steps you should take to install, configure, and run the kit.

### The Test Web Service

The TWS is a Web Service that simulates a typical database-driven Web service. TWS takes SOAP RPC encoded requests and returns a response. The response contains gibberish words of a length determined by a parameter of the request. TWS is a simple and easy choice to test application server platforms for performance, reliability, and scalability.

The kit implements TWS on a variety of application servers. For BEA WebLogic, I used BEA's integrated development environment named Workshop. Workshop's designers have figured out a clean way to present the myriad of options found in the SOAP and WSDL specifications. To see this in action, look at Figure 14–7.

Public methods in a Java class are turned into SOAP-based Web services automatically. For example, Figure 14–7 shows the `responder_rpc` class's single public method named `respond`. The right-most Properties pane shows the SOAP options for respond. Figure 14–7 shows how easy it is to change

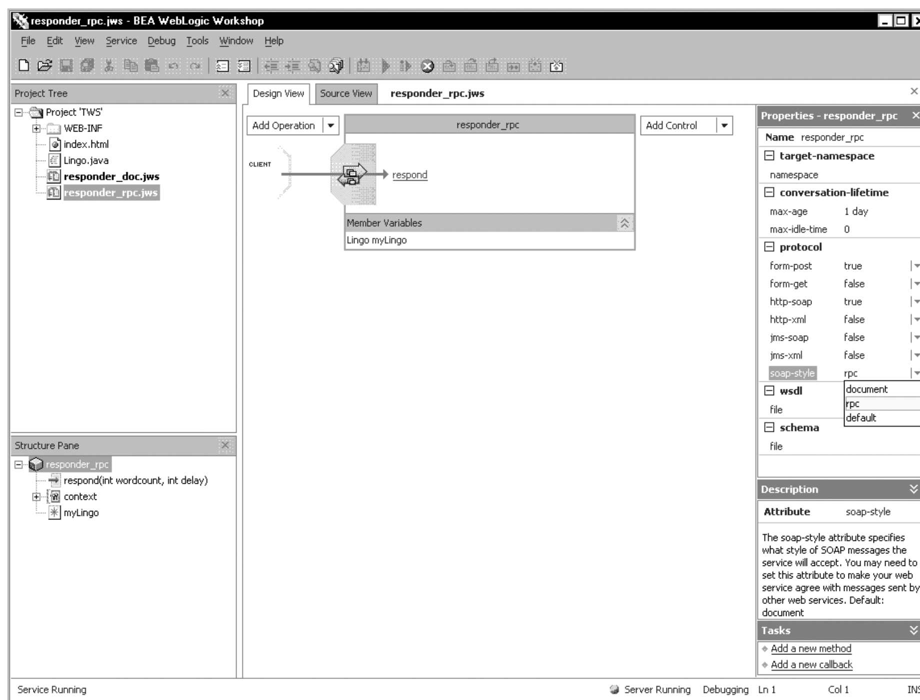


**Figure 14–7** Workshop makes the many SOAP options visually easy to adjust, including setting SOAP encoding styles between document-style and SOAP RPC-style.

SOAP encoding styles for the `respond` method. The choices are document, rpc, and default. WebLogic's default value chooses SOAP document-literal style encoding.

Choosing a SOAP encoding style in the Design View changes the JWS header in the Java code that is viewable in the Source View. JWS does the work for you. For example, the `respond` method in the Source View is declared by default as:

```
import weblogic.jws.control.JwsContext;

/**
* @jws:protocol form-get="false" soap-style="document"
*/

public String respond(int wordcount, int delay) { ...
```

This is enough for Workshop to package the deployment descriptors to tell the JWS engine to deploy a Web service that has a `respond` method that uses SOAP document-style encoding.

When we change the encoding style in the Design view to rpc, the Java code changes to:

```
import weblogic.jws.control.JwsContext;

/**
* @jws:protocol form-get="false" soap-style="rpc"
*/

public String respond(int wordcount, int delay) { ...
```

This does not appear to make much difference in the code; however, behind the scenes a lot is happening. And much of it impacts the scalability of the `respond` method. When this Java code is run on a JWS-compatible virtual machine—such as WebLogic Workshop—then the virtual machine automatically adds a SOAP proxy to the method so it will respond and reply to SOAP requests. The proxy handles all of the marshaling and unmarshaling of parameters to make the method call and respond with a response value.

To install and run TWS in WebLogic Workshop, run Workshop, open the TWS project, open the *responder_rpc.jws* file, and choose the Build and Debug command in the Debug drop-down menu. Workshop will compile the Java code into a WAR file, deploy the application as a Web service, and display a browser-based debug harness. Have some fun with the debug harness. It will let you type in calling parameters for TWS and immediately see the results.

As described in the Java code of TWS, the `respond` method has two parameters. `wordcount` is an integer value that tells TWS how to compose the result. TWS uses the `Lingo.java` object to compose a String value containing a bunch of gibberish words. The `wordcount` value determines the quantity of gibberish words. Consider this example where Lingo returns a phrase with 50 gibberish words:

```
from com.pushtotest.tool.util import Lingo
myLingo = Lingo()
print myLingo.getMessage( 50 )

Ia quantos quantos delorum quantos. Delorum ditchek quantos
so quantos ad. Lipsem so novus surbatton deplorem deplorem.
```

```
Delorum surbatton novus delorum. Lipsem ditchek so. Lipsem
quantos it infreteres ad deplorem ipsem surbatton.
Ca nmpus it ad ipsem novus it surbatton delorum. Campus
ipsem quantos novus sit ditchek ventes aqua ad ad aqua
ditchek. Novato aqua deplorem it infreteres infreteres quantos.
```

The `respond` method will return a response containing Lingo text of the size determined in the request. The `delay` parameter determines the number of milliseconds to sleep before responding to the request.

Once you have accomplished this task, please move on to the next section, where you will configure and run the PerfCheck test agent to drive the running Web service.

### PerfCheck

PerfCheck is an intelligent test agent written in TestMaker. Chapter 5 gives a thorough explanation of TestMaker. PerfCheck identifies the throughput of the TWS by making requests to TWS with increasing payload sizes and measuring the round-trip request/response time. PerfCheck averages the request/response times and presents a transactions-per-second result in the TestMaker output window.

PerfCheck issues requests to TWS for a predetermined amount of time, reports the TPS results metric, and then runs the test another time using a larger payload size. You should see the TPS value decrease as the payload size increases in accordance with the test results we found at PushToTest Labs and presented earlier in this chapter.

PerfCheck is organized among several script files to make the agent easier to understand. The following is a list and description of PerfCheck's contents.

- *properties.a*—Setup, configuration, and other parameters for the test.
- *master.a*—This is the main agent script to run. This script configures the test, stages the test, and records the results. The script displays the TPS and other performance metrics in the Output window.
- *tally.a*—Tally the results from the log file into TPS and other metrics.
- agents—A directory holding the scripts that implement the behavior needed to test the Web service.
- logs—A directory to hold the logged test result data.

Next, I take an in-depth look at these scripts and provide an explanation of how the test is constructed.

***properties.a*** The *properties.a* file contains these parameters to control the payload size and duration of the tests.

```
timetorun = 8 * 60 * 1000
```

PerfCheck runs in cycles. Each cycle increases the payload size of the TWS responses. The duration of the cycle is set here. For example, 8 tells PerfCheck to run each payload size for 8 minutes.

```
agentcount = 50
```

At the beginning of each cycle, PerfCheck instantiates a quantity of test agent threads that run concurrently. Each thread makes a request to TWS, logs the response time and result code, and then makes another request. The `agentcount` parameter controls the total number of concurrent threads to run.

```
payloadstart = 500
payloadinc = 1000
paycount = 4
```

PerfCheck uses these parameters to determine the payload size to begin the test with. The `payloadinc` value increases the payload size for each cycle. The `paycount` variable determines the number of cycles PerfCheck will run.

```
tws_host = "http://examples.pushtotest.com"
tws_port = 7001
tws_path = "TWS/responder_doc.jws"
```

PerfCheck uses these parameters to identify the location of TWS. The BEA Weblogic Workshop debug harness will show you the URL to TWS. The debug harness will also show you the WSDL description of the TWS to identify the location of TWS.

***master.a*** The *master.a* script stages the test by loading the property values, instantiating the test threads, and then running the *tally.a* script. To run a test, run the *master.a* script. The results will appear in the TestMaker output panel. A closer look at *master.a* shows how it accomplishes these tasks. Here is *master.a* in its entirety, followed by an in-depth explanation of the *master.a* script.

```
# master.a

from java.lang import Thread
import java
import time
from java.util import Date
from com.pushtotest.tool.logger import simplelogger
import sys
import thread

print "Configuring to run the test."

exec open( scriptpath + "Properties.a" ).read()

payloadsize = payloadstart

for c in range( paycount ):
    running = 1
    up_count = 0
    record = 0
    tick = 0
    errortick = 0

    first_time = Date().time

    logtarget = resultspath + "results_" + localname \
    + str(c) + ".txt"

    try:
        log = simplelogger.getInstance()
        log.setFileName( logtarget )
    except java.lang.Exception, ex:
        print "Could not start logging."
        sys.exit()

    total = 0
    for a in agents:
        total += a[1]

    if total>100:
        print "Warning: The total number of agents"
        print "         in the Properties.a file is"
        print "         more than 100%."

    for at in agents:
```

```
        agenttype = at[0]

        agent_count = int ( ( float( at[1] ) / \
        float( total ) ) * agentcount )

        for i in range( agent_count ):
            exec open( scriptpath + "agents/" + \
            agenttype + ".a" ).read()

            time.sleep( startupdelay )

    print "Threads started, we're testing!"

    record = 1

    start_time = Date().time
    end_time = start_time + timetorun

    while Date().time < end_time:
        time.sleep( 1 )

    record = 0

    recording_time = Date().time - start_time

    running = 0     # Time to stop

    log.closeLogFile()

    close_time = Date().time
    close_end_time = close_time + maxwait

    while ( Date().time < close_end_time ) and \
    ( up_count > 0 ):
        time.sleep( 1 )

    duration = Date().time - start_time

    exec open( scriptpath + "Tally.a" ).read()

    payloadsize += payloadinc
```

The *master.a* script is fairly straightforward and procedural. It starts the test, runs the test threads, then runs the *tally.a* script and cleans up the test.

Next I cover the *master.a* script step-by-step. Chapters 5, 7, and 11 feature other test agents that follow a similar design pattern as *master.a*.

```
from java.lang import Thread
import java
import time
from java.util import Date
from com.pushtotest.tool.logger import simplelogger
import sys
import thread
```

The *master.a* script begins with a few import statements. TestMaker provides the Jython scripting language. Jython is the popular Python programming language implemented 100% in Java. Details on Jython are in Chapter 5. The import statements identify the Python and Java objects Jython will use to perform the test. For example, later in this agent we will use the Thread object. This is Java's Thread object for running objects concurrently.

```
exec open( scriptpath + "Properties.a" ).read()
```

The `exec` command runs the script commands found in the *properties.a* script. The `scriptpath` variable is a TestMaker system variable that contains the path to the currently running script.

```
payloadsize = payloadstart

for c in range( paycount ):
```

*master.a* will cycle through tests with increasing payload sizes. The `payloadsize` variable controls the size of the payload. The `payloadsize` variable initializes to a value set from the *properties.a* script.

```
    running = 1
    up_count = 0
    record = 0
    tick = 0
    errortick = 0
```

These variables are used to control the concurrently running test threads we are about to create. `running` is a flag used to tell the test agents when to run and when to stop. `up_count` counts the number of agent threads that are up and running. `record` is a flag that tells the running test threads when to

start logging results data. `tick` is used to keep track of the total number of completed transactions. `errortick` keeps track of transactions that ended in an error condition.

```
first_time = Date().time
```

`first_time` keeps track of the time the test started. This will be used later when the *tally.a* script reports the results.

```
logtarget = resultspath + "results_" + localname \
+ str(c) + ".txt"

try:
    log = simplelogger.getInstance()
    log.setFileName( logtarget )
except java.lang.Exception, ex:
    print "Could not start logging."
    sys.exit()
```

The *master.a* script uses TestMaker's log handler to output the test results to a log file. The test threads will use this log handler.

```
total = 0
for a in agents:
    total += a[1]

for at in agents:

    agenttype = at[0]

    agent_count = int ( ( float( at[1] ) / \
    float( total ) ) * agentcount )

    for i in range( agent_count ):
        exec open( scriptpath + "agents/" + \
        agenttype + ".a" ).read()
```

This code calculates how many of each type of thread to run based on the percentage value set in *properties.a*. For the purposes of this test the *properties.a* script uses only a single type of test thread. However, you may want to see if using a combination of SOAP encoding styles impacts scalability and performance in your environment.

The `agents` variable is a Jython list object that contains a set of tuples. A tuple here holds two values, the test thread name, and the percentage mix:

```
[ "wlw_agent_rpc_lit", 75 ]
[ "wlw_agent_docstyle", 25 ]
```

The first value of each tuple identifies by name the user archetype to use for a test agent and the second value indicates the percentage mix of the type of test agent to use when testing. For example, `["wlw_agent_rpc_lit", 75]` indicates that 75% of the total number of concurrently running agents will be using the `wlw_agent_rpc_lit` test thread behavior.

```
            time.sleep( startupdelay )
```

After each agent thread instantiation, the script uses the sleep method to give some time for the thread to bring up a client connection before starting the next one. Without this delay, the threads would start simultaneously and may overrun the environment's resources to handle instantiating all the threads. As a result, one or more threads might time out before connecting to the host.

```
    record = 1
```

All the threads are instantiated so changing the value of the record global variable tells the agents to begin logging results to the `log` object.

```
    start_time = Date().time
    end_time = start_time + timetorun

    while Date().time < end_time:
        time.sleep( 1 )
```

Wait around until the amount of time for the test elapses. While the *master.a* script sleeps, the instantiated test threads communicate with the target host.

```
    record = 0

    recording_time = Date().time - start_time

    running = 0     # Time to stop
```

```
    log.closeLogFile()

    close_time = Date().time
    close_end_time = close_time + maxwait
```

The test period is over. Setting the record global variable tells the running agents to stop logging results.

```
    while ( Date().time < close_end_time ) and \
    ( up_count > 0 ):
        time.sleep( 1 )
```

The *master.a* waits around while the test threads finish and exit. The *master.a* script then calls the *tally.a* script and exits.

**test threads** The test threads are defined in the agents directory. All of the test threads follow the same design pattern. Rather than showing the entire thread, I will show how each thread uses TestMaker to use the different SOAP encoding styles.

The *wlw_agent_rpc.a* script implements the code necessary to make SOAP RPC-encoded requests to the TWS. Here is a snippet of the script showing how to make the SOAP RPC request to the Web service:

```
protocol = ProtocolHandler.getProtocol("soap")
body = SOAPBody()
protocol.setBody(body)

protocol.setUrl( endpoint )

body.setTarget( "http://www.openuri.org/" )
body.setMethod( "respond" )

body.addParameter("wordcount", Integer, payloadsize, None)
body.addParameter("delay", Integer, 0, None )

response = protocol.connect()
```

The `body.addparameter()` method provides a powerful way to send objects to the Web service. The format allows you to add any object type that is on the TestMaker Java classpath. In this example we are sending two Integer data types. The first is the `wordcount` value and the second is the `delay` value.

Compare the relatively easy SOAP RPC encoded request to the next snippet of code. The following code is from the *wlw_agent_docstyle.a* script that implements the code necessary to make a SOAP document-literal encoded request to the Web service. As we saw earlier in this chapter, document-literal encoding receives an XML data structure already well formed and ready to send to the host.

```
protocol = ProtocolHandler.getProtocol("soap")
body = SOAPBody()
protocol.setBody(body)

protocol.setUrl( endpoint )

body.setMethod( "respond" )

ns = "http://www.openuri.org/"

elRespond = Element( "respond", ns )

ellen = Element( "wordcount", ns )
ellen.addContent( payloadsize + "" )
elRespond.addContent( ellen )

eldel = Element( "delay", ns )
eldel.addContent( "0" )
elRespond.addContent( eldel )

doc = Document( elRespond )

body.setDocument( doc )

response = protocol.connect()
```

This snippet of test thread script creates an XML document using the JDOM API that described in Chapter 7. The request document looks like this when built:

```
<s:Envelope
   xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <respond xmlns="urn:responder_msg">
      <delay xmlns="urn:responder_msg">25</delay>
      <length xmlns="urn:responder_msg">75</length>
```

```
    </respond>
  </s:Body>
</s:Envelope>
```

We use the JDOM APIs to build the XML request in a DOM object and then pass the DOM object to TOOL to make the SOAP call to the server. TestMaker uses the `namespace` parameter to identify the designation Web service name. The endpoint gets us to the right server, the namespace of the first element gets us to the right Web service.

Finally, the last snippet of code shows the *wlw_agent_rpc_lit.a* scripts use of SOAP RPC-literal encoding. In this snippet we use the same JDOM APIs to construct the XML document. However, we will use the SOAP RPC encoding commands to pass the XML document as a single element.

```
protocol = ProtocolHandler.getProtocol("soap")
body = SOAPBody()
protocol.setBody(body)

protocol.setUrl( endpoint )

protocol.setEncoding( Constants.NS_URI_LITERAL_XML )

body.setTarget( "http://tempuri.org/responder_doc" )
body.setMethod( "respond" )

elOne = Element( "respond" )

elDelay = Element( "delay" )
elDelay.addContent( "12" )
elOne.addContent( elDelay )

elLength = Element( "length" )
elLength.addContent( "25" )
elOne.addContent( elLength )

domo = DOMOutputter()

body.addParameter( "inDoc", w3c_element, \
domo.output( elOne ), Constants.NS_URI_LITERAL_XML )
```

This code snippet is similar to the previous snippet that used SOAP document-literal encoding. This code used the SOAP RPC encoding calls with the following difference: `setEncoding( Constants.NS_URI_LITERAL_XML)`

tells TestMaker that the parameters of the request and response will use document-literal encoding. Additionally, the `addParameter` method passed in a `w3c_element` type of object containing the XML data.

As you can see the kit provides a flexible and powerful way to test the popular SOAP encoding styles for scalability and performance. To start a test, open TestMaker, navigate to the kit test agent scripts, and open and run the *master.a* script. The script displays the TPS and other performance metrics in the Output window.

## Some Other Things We Found

While building and running the test environment for Elsevier, I noticed a few things worthy of mentioning here. First, each application server platform I worked with has its own implementation of a SOAP stack. Some even shipped with more than one stack. For example, BEA WebLogic server comes with a SOAP stack that implements the JAX-RPC API and one that implements the JWS APIs. IBM WebSphere comes with the DOM-based Apache SOAP library, but also has the SAX-based Apache AXIS library. I would love the opportunity to test the differences between all of these in some future project.

I found that while WSDL did a fair job at describing the interface to a SOAP service, many times WSDL was not complete. I needed to use a network monitor to see actually what values were moved over the HTTP transport.

I also noted that while building the test environment for Elsevier, BEA and IBM announced major new versions of their Web service application servers. Each new version changed their SOAP stack implementation with positive and negative impact on performance and scalability.

Reliability and performance of SOAP application servers is a moving target. This further reinforced the lesson that a test environment is necessary to check reliability and performance now and as new implementations become available.

With the test environment complete, Elsevier Science now has the means to understand reliability, scalability, and functionality of its systems. The test agent technology will be employed as new builds of the Elsevier software and the Web service application servers become available. The test environment is planned also to provide ongoing tests for regression, functional tests, and as a proof-of-service monitor.

## Summary

In this chapter, we found that software developers have many choices for building Web service systems: SOAP encoding styles, Web service development tools, and application servers. This chapter presented the results of an investigation that shows how each choice immediately impacts scalability and reliability.

Elsevier adopted SOAP as their standard way to build their next-generation content aggregation system. We saw how Elsevier developed a new methodology and test environment to check various SOAP implementations, including application servers, SOAP stacks, and utilities, for scalability and performance.