Listing 15.29 *(continued)*

```
 <hr />
</xsl:template>

<xsl:template match="shiplist/ship">
 <h4><xsl:value-of select="name"/></h4>
 <p>Ship type is <xsl:value-of select="@type"/>,
    registered in <xsl:value-of select="registry"/></p>
</xsl:template>

</xsl:stylesheet>
```

When configured as the preceding, and called as `http://localhost/axkit/captains.xml`, `Apache::AxKit::Language::LibXSLT` will be used to render the HTML output to the browser.

15.16. Creating a SOAP Server

You want to create a SOAP server that allows you to remotely call perl procedures.

Technique

Install and configure the `Apache::SOAP` module, available as part of the `SOAP::Lite` distribution on CPAN.

The following directives, when added to your `httpd.conf`, allow you to remotely call methods in the `HalfLife::QueryServer` class, which is defined in the discussion. Accessing your own modules is as simple as changing the module name in this example.

```
PerlModule Apache::SOAP

<Location /game-query>
  SetHandler perl-script
  PerlModule Apache::SOAP

  PerlSetVar dispatch_to 'HalfLife::QueryServer'
</Location>
```

Comments

SOAP (Simple Object Access Protocol) is an XML-based protocol that defines an RPC-like architecture between a client and server for use over any transmission medium. This allows simple effective communication between applications written in different languages running on different platforms.

The SOAP protocol specifies the message format as well as serialization rules, which allows for maintaining data integrity even over mediums that do not have built-in state mechanisms, like HTTP. Currently, the only medium that has widespread SOAP support is HTTP, which is fortunate for us as Apache/mod_perl developers. `SOAP::Lite` is a Perl toolkit that takes care of all the complex nastiness of actually implementing the SOAP protocol over HTTP and allows Perl programs to focus on simply creating our classes, methods, and objects, which is also fortunate—not many other languages have SOAP libraries that are as easy to use or as intuitive as `SOAP::Lite`.

In support of the server side of the SOAP protocol, `SOAP::Lite` provides the `Apache::SOAP` module, which hooks into mod_perl to provide an interface to the Perl modules resident on your server. To use `Apache::SOAP`, you have to deviate from the standard `SOAP::Lite` installation and be certain to add mod_perl support to the list of installed modules. This is done during the initial configuration; be sure to answer `no` to the default configuration and `yes` at the Apache/mod_perl prompt. Alternatively, you can use the shortcut arguments when creating the `Makefile`.

```
$ perl Makefile.PL --HTTP-Apache --noprompt
```

See the `SOAP::Lite` documentation for more information about installation procedures.

The example we chose to showcase `Apache::SOAP` is more entertaining than practical, but it does illustrate the simple elegance of SOAP. We start with a very basic module, `HalfLife::QueryServer`, which provides a partial interface into the server network protocol for the popular computer multiplayer game Half-Life. It allows you to query a specific Half-Life server for some detailed information, such as the OS platform and current game map. The specifications of the Half-Life network protocol can be found "unofficially" throughout the Web, whereas an active game server can be tracked down using a game server monitor such as `aGSM`.

Place the following code into a file named `HalfLife/QueryServer.pm`. Be sure this file is found within mod_perl's library path.

```
package HalfLife::QueryServer;

use IO::Socket;
use NetPacket::IP;
use NetPacket::UDP;

use strict;

sub new {

  my $self  = shift;
  my $class = ref($self) || $self;

  my ($server, $port) = @_;

  return bless { _ip   => $server,
                 _port => $port || 27015
  }, $class;
}

sub ping {

  my $self   = shift;

  my $server = IO::Socket::INET->new(PeerAddr => $self->{_ip},
                                     PeerPort => $self->{_port},
                                     Proto    => 'udp',
                                     Timeout  => 5,
                                     Type     => SOCK_DGRAM)
    or die "could't open socket: $!";

  $server->send("\xFF\xFF\xFF\xFFdetails\x00");
  $server->recv(my $packet, 1024);

  return $self->_parse_response($packet);
}

sub remotequery {
  # Query the server and return some results, all in one command.

  my $self = new(@_);

  $self->ping();
```

```
    return [$self->{_os}, $self->{_type},
            $self->{_map}, $self->{_description}];
  }

  sub ip {

    my $self = shift;

    return $self->{_ip} unless @_;

    return $self->{_ip} = shift;
  }

  sub port {

    my $self = shift;

    return $self->{_port} unless @_;

    return $self->{_port} = shift;
  }

  sub os {
    return shift->{_os};
  }

  sub type {
    return shift->{_type};
  }

  sub map {
    return shift->{_map};
  }

  sub description {
    return shift->{_description};
  }

  sub _parse_response {

    my ($self, $packet) = @_;
  my $response = NetPacket::UDP->decode($packet);

  my ($address, $server, $map, $directory, $description,
      $decode_me, $info, $ftp, $version, $bytes,
      $servermod, $customclient) = split /\0/, $response->{data};

  my ($active, $max, $proto, $type, $os, $password, $mod) =
    map { ord(substr($decode_me,$_,1)) } (0 .. 6);

  $self->{_os} = $os eq 'l' ? 'linux' : 'windows';
  $self->{_type} = $type eq 'd' ? 'dedicated' : 'listener';
  $self->{_map} = $map;
  $self->{_description} = $description;
  $self->{_server} = $server;
}
1;
```

The methods of the `HalfLife::QueryServer` class should be pretty self-explanatory. A `HalfLife::QueryServer` object is created with the `new()` constructor, which is followed by a call to `ping()` that actually initiates the query and parses the response. After that, we can access attributes of the game server using the `os()`, `map()`, and other methods. Tracing the `ord()` and `IO::Socket` calls, as well as creating accessor methods for the missing server attributes, is an exercise left to the reader. Although the code is overly object-oriented for such an easy task, it was written this way on purpose to illustrate the object-oriented nature of SOAP and the realm of possibilities `SOAP::Lite` opens up.

In order for us to use our (somewhat contrived) example, we need to turn our standard mod_perl server into a SOAP server. For this, we use the `Apache::SOAP` module and configure Apache with the directives found in the solution to this recipe. `Apache::SOAP` is a simple mod_perl handler that implements the SOAP protocol over HTTP. It only requires a single configuration parameter, the `PerlSetVar dispatch_to`, which specifies the class to which the incoming request to `/game-query` belongs. Although we chose a single module for our configuration, `dispatch_to` can also point to an absolute path on the server, which open up the `<Location game-query>` to any number of different Perl classes similar to the `use lib` pragma.

When a request comes in to `/game-query`, `Apache::SOAP` handles the request, translating the incoming HTTP message into calls to the `HalfLife::QueryServer` class and sending a properly formatted SOAP response back. The key to making SOAP work over HTTP is its use of the message body for passing object data between the client and server. SOAP is unique in the HTTP world in that it uses the HTTP message body for both the request and response phases of the transaction. Thus, a typical request cycle might look something like

```
POST /game-query HTTP/1.0
Accept: text/xml, multipart/*
Content-Length: 535
Content-Type: text/xml; charset=utf-8
Host: www.example.com
SOAPAction: "http://www.example.com/HalfLife/QueryServer#new"
User-Agent: SOAP::Lite/Perl/0.51

<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"><SOAP-ENV:Body><namesp1:new
xmlns:namesp1="http://www.example.com/HalfLife/QueryServer"><c-gensym3
xsi:type="xsd:string">10.3.4.200</c-gensym3></namesp1:new></SOAP-
ENV:Body></SOAP-ENV:Envelope>

HTTP/1.0 200 OK
Content-Length: 731
Content-Type: text/xml; charset=utf-8
SOAPServer: SOAP::Lite/Perl/0.51
Connection: close

<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:namesp10="http://www.exam-
ple.com/HalfLife/QueryServer"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"><SOAP-
ENV:Body><namesp11:newResponse
xmlns:namesp11="http://www.example.com/HalfLife/QueryServer"><HalfLife__QuerySer
ver xsi:type="namesp10:HalfLife__QueryServer"><_ip
xsi:type="xsd:string">10.3.4.200</_ip><_port
xsi:type="xsd:int">27015</_port></HalfLife__QueryServer></namesp11:newResponse><
/SOAP-ENV:Body></SOAP-ENV:Envelope>
```

Note the `SOAPAction` request header, which specifies the class and method to call, and the `10.3.4.200` values in both the request and response message body, which is the data used to construct our new `HalfLife::QueryServer` object.

Let's put our flashy new mod_perl SOAP server to work so we can solidify the rather abstract concepts here. Here is an implementation of a SOAP client that connects to the server we previously described. Put the following code into a Perl script (for example `querysoap.pl`) and execute it. It sends a SOAP request to the server and prints the output from the `os()` and `map()` methods.

Listing 15.30 `querysoap.pl`

```
#!/usr/bin/perl

use SOAP::Lite +autodispatch =>
  uri => 'http://www.example.com/HalfLife/QueryServer',
  proxy => 'http://www.example.com/game-query';

use strict;

my $hl = HalfLife::QueryServer->new('10.3.4.200');

$hl->ping;

print $hl->os, "\n";
print $hl->map, "\n";
```

Take a moment to digest that relatively plain bit of code. The first thing to note is the rather odd use of the `use` pragma. The arguments here establish the behavior of the rest of the Perl code. The most important parameter is `proxy`, which determines the location of the SOAP server. The `uri` parameter specifies the namespace of the SOAP service on the proxy. This goes back to the `PerlSetVar dispatch_to` configuration where we could specify a number of different classes to make available through our SOAP server. Keep in mind that although the `uri` argument looks like a URL, it is merely a distinct namespace—the only thing that matters is the path component of the URI.

What really makes `SOAP::Lite` a powerful programming tool is the `autodispatch` option, which makes all the method calls in the SOAP client dispatch to the proper SOAP service. Did you notice the lack of a `use HalfLife::QueryServer;` call? SOAP and `SOAP::Lite` together are almost eerie in the way they provide true encapsulation: Your class definitions, class methods, and object methods reside entirely on the SOAP server. So, not only is the implementation of the entire API for our class hidden from the SOAP client, it does not even have to be in the same language. Here is a SOAP client written in Java that can access our mod_perl SOAP server, and the underlying Perl classes and methods.

Listing 15.31 GetHalfLife.java

```
// Simple command line tool to retrieve HalfLife Information

import java.io.*;
import java.util.*;
import java.net.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import javax.xml.parsers.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.rpc.*;
import org.apache.soap.transport.http.SOAPHTTPConnection;

public class GetHalfLife {
    public static final String DEFAULT_SERVICE_URL =
        "http://www.example.com/game-query";

    public static void main(String[] args) throws Exception {
        String serviceURL     = DEFAULT_SERVICE_URL;

        URL url = new URL(serviceURL);

        // create the transport and set parameters
        SOAPHTTPConnection st = new SOAPHTTPConnection();

        // build the call.
        Call call = new Call();
        call.setSOAPTransport(st);
        call.setTargetObjectURI("urn:HalfLife/QueryServer");
        call.setMethodName("remotequery");
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

        Vector params = new Vector();
        params.addElement(new Parameter("server", String.class,
                                        "10.3.4.200", null));
        call.setParams(params);

        // Send request to Halflife SOAP Server
```

Listing 15.31 *(continued)*

```
        System.err.println("Invoking Halflife service at: ");
        System.err.println("\t" + serviceURL) ;

        Response resp;
        try {
            resp = call.invoke(url, "");
        } catch(SOAPException e) {
            System.err.println("Caught SOAPException (" +
                               e.getFaultCode () + "): " +
                               e.getMessage ());
            return;
        }

        // check response
        if (!resp.generatedFault()) {
            Parameter ret     = resp.getReturnValue();
            Object    value   = ret.getValue();
            String[]  results = (String[])value;

            // Print out the returned array of information.

            for (int i = 0; i < results.length; i++)
                    System.out.println(results[i] );

        } else {
            Fault fault = resp.getFault();
            System.err.println("Generated fault: ");
            System.out.println("  Fault Code   = " + fault.getFaultCode());
            System.out.println("  Fault String = " + fault.getFaultString());
        }
    }
}
```

The preceding Java code uses functionality provided by the Apache-SOAP project available from `http://xml.apache.org/soap/`. Put the preceding code into a file named `GetHalfLife.java`. The following commands will compile and execute the code.

```
$ javac GetHalfLife.java
$ java GetHalfLife
Invoking Halflife service at:
 http://www.example.com/game-query
windows
listener
badlands
Team Fortress Classic
```

The preceding Java code shows just how clever and simple `SOAP::Lite` truly is. With it we can create SOAP clients and SOAP servers in a few lines of configuration and code.