Gründlich eingeseift

SOAP Web Services mit PHP

Web Services und speziell das Simple Object Access Protocol SOAP [1] sind momentan in aller Munde. Mit SOAP gelingt die Anbindung an größere Softwareeinheiten wie Billing Systeme oder Application Server. Schaut man hinter den Hype der Web Services, so findet man einen portablen, in vielen Sprachen und Plattformen einsetzbaren Standard zur Interoperabilität. Auch für PHP gibt es eine Reihe von SOAP-Implementierungen, von denen hier eine vorgestellt und in einem praktischen Beispiel eingesetzt werden soll.

Dr. Volker M. Göbbels

Unter dem Stichwort Interoperabilität versteht man die Möglichkeit, von einem Rechner (Client) aus Aktionen, Programme oder Routinen auf einem anderen Rechner (Server) aufzurufen und Ergebnisse dieser Aktion zurückgeliefert zu bekommen. In der Unix-Welt ist diese Technik schon lange unter dem Stichwort Remote Procedure Calls oder RPC bekannt. Ein Analogon hierzu stellt XML-RPC dar. Ein deutlich modernerer Standard mit ähnlicher Funktionalität ist SOAP. Im Widerspruch zu seinem Namen beschäftigt sich SOAP aber in keiner Weise mit Objekten im Sinne der OOP.

Das Publizieren von Applikationsdiensten per Webschnittstelle ist allerdings eine neue und bisher nur ansatzweise genutzte Methode, Rechner irgendwo auf der Welt flexibel zu verbinden. Die Nutzung des Web als Transportbasis hat den Vorteil, dass der Datenverkehr ungehindert über den auf den meisten Firewalls und Gateways freigeschalteten Port 80 laufen kann. Änderungen an der Infrastruktur sind daher nicht nötig. Die Daten werden in XML ja zudem im ASCII-Format übertragen.

Transportprotokolle

SOAP Messages können prinzipiell über jedes Netzwerkprotokoll übertragen werden, welches Texte transportieren kann. Eingesetzt werden allerdings lediglich HTTP und SMTP. SMTP hat jedoch den Nachteil, asynchron zu arbeiten, d.h., die Nachricht kommt genau dann beim SOAP Server an, wenn dieser sie vom Mail Server abholt. Daher hat sich HTTP als Standard-Transportprotokoll für SOAP Messages durchgesetzt.

Zusätzlich bietet HTTP über die HTTP Response die Möglichkeit, aus der per Standard als Einwegübertragung definierten SOAP Message durch Rückgabe einer HTTP Response, die wieder eine SOAP Message enthält, ein Zweiwegeprotokoll zu machen. Die Verwendung eines Webservers (in unserem Fall der Apache) ermöglicht auch den Einsatz von PHP im SOAP Serverpart, ohne selbst einen Socketserver schreiben zu müssen. Das PHP SOAP Serverskript liegt einfach im document tree des Webservers und ein Aufruf der URL dieses Skripts setzt den SOAP Server für diese eine Abfrage in Gang. Etwas komplizierter wird das Szenario dadurch, dass der beim SOAP Server anfragende Rechner in unserem Fall selbst ein Webserver bzw. ein PHP Skript ist (siehe Abb. 1).

Der doppelte Indianer

Zum Experimentieren mit SOAP benötigt man aber nun nicht gleich zwei getrennte Webserver. Client- und Serverfunktionalität können mit einem Server bedient werden. Um den SOAP Client- und Serverteil deutlicher vom Webtraffic zu trennen und ein „Abhören“ des Transports im Netz zu ermöglichen, lassen wir die SOAP Kommunikation nicht über Port 80 sondern über den privaten Port 20080 laufen. Die hierzu nötige Konfiguration des Apache zeigt das folgende Lis-

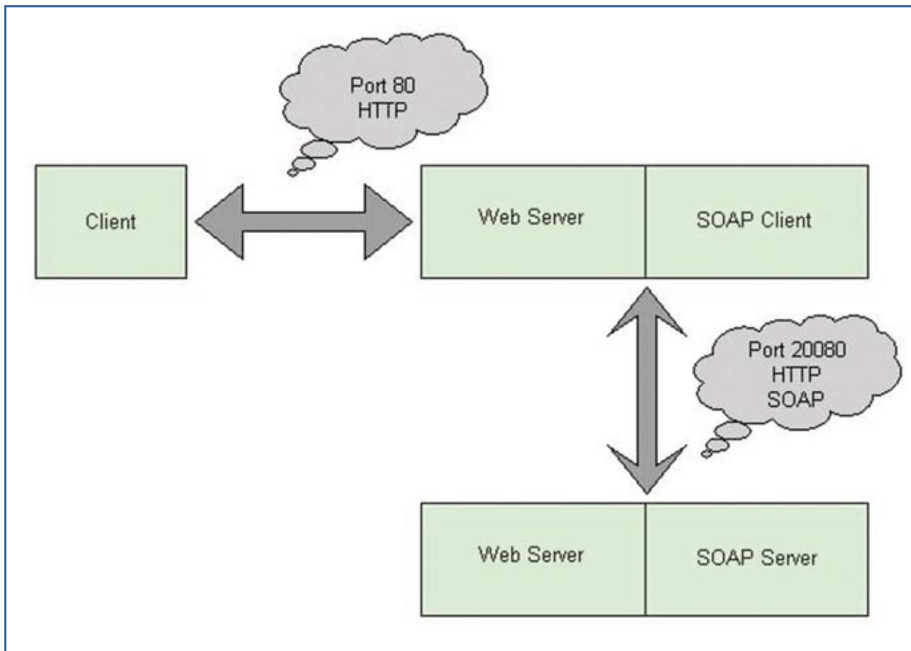

Abb. 1: Ein-Server-Szenario für SOAP Anwendungen

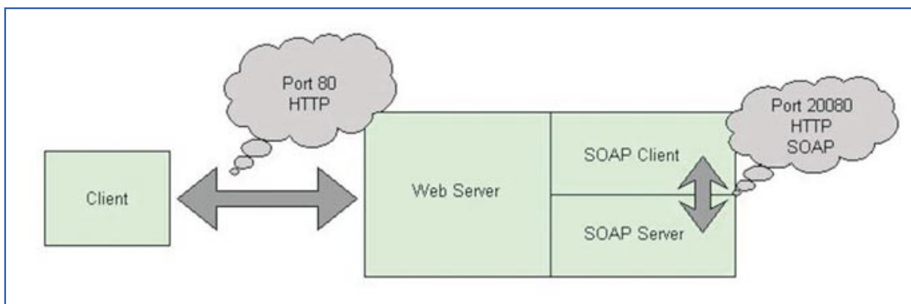

Abb. 2: Ein-Server-Szenario zum Test von SOAP Applikationen

ting, welches Ausschnitte aus der *httpd.conf* enthält:

```
ServerRoot „/opt/prj/cw"
...
Listen 80
Listen 20080
...
DocumentRoot „/opt/prj/cw/htdocs"
...
<VirtualHost 192.168.0.99:20080>
 DocumentRoot /opt/prj/cw/soap
 <ifModule mod_dir.c>
  DirectoryIndex index.html index.php
 </ifModule>
</VirtualHost>
```

Startet man den so konfigurierten Apache, zeigt ein Blick in die Liste der offenen Ports die korrekte Arbeitsweise:

```
paradiso:/usr/local/web # lsof -i
...
httpd    766    root    16u    IPv4
6006     TCP *.soap (LISTEN)
httpd    766    root    17u    IPv4
6007     TCP *.http (LISTEN)
...
```

Diese „port based virtual host" Konfiguration lässt den Apache auf zwei Ports horchen: Webport 80 zeigt Dokumente ab dem document root */opt/prj/cw/htdocs*, Port 20080 verweist auf ein document root in */opt/prj/cw/soap*. Hier liegt das später noch vorzustellende Serverskript *SOAP_SERVER.php*. In dieser Konfiguration wird kein *NameVirtualHost* definiert. Die Struktur dieses Setups zeigt Abbildung 2.

In */opt/prj/cw/soap* liegen auch die Klassen der SOAP Implementierung SOAPx4 von Dietrich Ayala [2], *class.soap_server.php* und *class.soap_client.php*. Dieses Verzeichnis sollte in *php.ini* daher auch im *include_path* aufgeführt werden.

Anatomie eine SOAP Message

Auch wenn sich eine SOAP Implementierung um das Ein- und Auspacken von Funktionsaufruf und Antwort kümmert, sollte man über den grundsätzlichen Aufbau einer SOAP Message informiert sein. Insgesamt gibt es drei Typen von Messages, die alle den in Abbildung 3 dargestellten Aufbau besitzen.

Unterschieden werden SOAP Aufrufe, Responses und Fault Messages. Allen gemeinsam ist der Aufbau bestehend aus einem Envelope, einem optionalen Header und einem Body, der die eigentliche Nachricht und ihre Parameter, auch *payload* genannt, enthält. Wie bei einem XML-basierten Protokoll nicht anders zu erwarten, stellen alle drei Typen valide XML Dokumente dar, unterliegen jedoch ein paar Einschränkungen:

- Das Rootelement des Dokuments heißt immer Envelope.
- Das Dokument darf keine Schemadefinition enthalten, weder in Form einer DTD noch als XSchema. Dies würde auch wenig Sinn ergeben, da die Syntax einer SOAP Message genau festgelegt ist.
- Der Namespace für Envelope, Header und Body ist immer SOAP-ENV.

Da ein konkretes Beispiel oft mehr sagt als tausend Worte, sehen Sie in Listing 1eine einfache Message aus unserem kommenden Fallbeispiel, in dem eine Adressdatenbank in MySQL anhand eines Nachnamens abgefragt werden soll.

Dieser Aufruf enthält einen Envelope, der eine Reihe von Namespaces definiert. Die wichtigsten hiervon sind:

- *xsd* (XML Schemadefinition) und *xsi* (XML Schemadefinition Instance) zur Definition der verwendeten XML Datentypen.
- *SOAP-ENC* für das Encoding, das heißt das Verpacken oder Serialisieren der Datenstrukturen.

- *SOAP-ENV* als Namespace für die SOAP Message selbst.
- *ns6* als Namespace der eigentlichen Funktionsaufrufe und Parameter.

In diesem speziellen Fall wird die Routine *getAdr* mit genau einem Parameter aufgerufen. Dieser Parameter ist ein Array (*struct*) mit dem Namen des PHP-Arrays (*query*), mit dem der Aufruf erfolgte, und enthält lediglich ein Element, nämlich *name=„huber“*.

Diese Message enthält keinen Header. Ein Header dient dazu, zusätzliche Daten mit dem eigentlichen Aufruf zu verschicken. Typische Beispiele sind Transaktionsdaten, wie sie ein Beispiel aus der SOAP Recommendation des W3C [1] zeigt:

```
<SOAP-ENV:Header>
 <t:Transaction xmlns:t="some-URI" SOAP-ENV:mustUnder-
stand="1">
  5
 </t:Transaction>
</SOAP-ENV:Header>
```

Das Attribut *mustUnderstand=1* besagt, dass der Empfänger dieses Element namens *Transaction* mit dem Wert *5* verstehen muss. Kann er das nicht, muss er eine SOAP Fehlermeldung zurückliefern und darf den Aufruf nicht weiter ausführen.

Das einzige weitere standardisierte Attribut zu einem Headerelement ist *actor*. Sollte eine SOAP Message nicht direkt an den ausführenden Rechner (den so genannten Endpoint) geschickt werden, sondern einen oder mehrere SOAP-Proxies durchlaufen, also Rechner, die SOAP Messages lesen und interpretieren bzw. weiterleiten können, kann mit dem Actor der tatsächliche Endpunkt definiert werden. In den meisten Standardfällen wird man SOAP Header nicht benötigen.

Die Antwort auf unsere Anfrage könnte aussehen wie in Listing 2.

Die Antwort besteht laut Standard aus einem einzigen Element mit dem Namen der aufgerufenen Routine (*getAdr*) und angehängtem „Response“. Darin befinden sich alle zurückgegebenen Daten, in diesem Fall wieder ein Array mit dem Namen der antwortenden Routine. Darin enthalten sind die eigentlichen Daten.

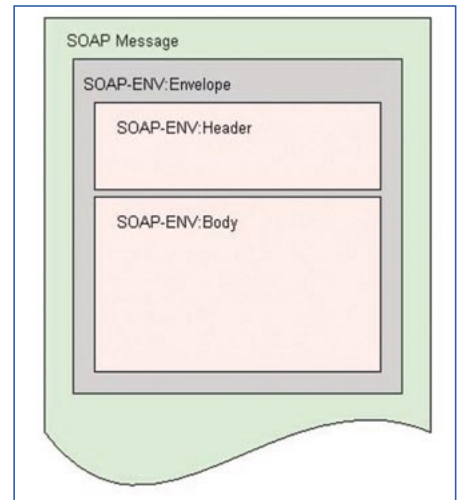

Abb. 3: Allgemeiner Aufbau einer SOAP Message

Der letzte Typ SOAP Messages sind schließlich die Fault Messages. In diesen wird eine Fehlermeldung vom SOAP Server auf den Client transportiert. Ein typisches Beispiel aus der SOAP Recommendation finden Sie in Listing 3.

Fault Messages haben wieder genau ein Element zum Inhalt. Dieses hat den Namen *Fault* und kann eine Reihe Subelemente enthalten:

Anzeige

- *faultcode* enthält einen maschinell verarbeitbaren Fehlercode und ist zwingend vorgeschrieben. Der SOAP Standard definiert eine kleine Anzahl dieser Codes (Client, Server, Version-Mismatch, MustUnderstand). Diese Faultcodes gehören zum Namespace *SOAP-ENV* und können durch eigene „Subklassen" von Fehlern erweitert werden, indem die eigene Fehlerbezeichnung mit einem Punkt angehängt wird, z.B. *Server.NotFound*.
- *faultstring* enthält eine menschenlesbare Fehlermeldung und muss angegeben werden.
- *faultactor* bezeichnet den Ort bzw. die URL, analog zum *actor* Attribut, an dem der Fehler aufgetreten ist. SOAP Applikationen, die nicht der Endpunkt einer SOAP Message sind, müssen einen *faultactor* mit dem eigentlichen Ursprung des Fehlers anlegen. Der tatsächliche Endpunkt der Message, an dem der Fehler aufgetreten ist, kann einen solchen *faultactor* aufnehmen.
- *detail* ist dazu gedacht, Inhalte zu transportieren, die durch einen Fehler bei der Ausführung der Applikation auf dem Server erzeugt wurden, also den Inhalt des Body betreffen. In diesem Fall ist das *detail* Element Pflicht. Sollte der Fehler durch das SOAP Framework auf dem Server entstanden sein oder durch den Header der Message, so darf kein *default* Element angegeben werden.

Hier werden Sie geholfen

Nachdem wir nun schon gesehen haben, wie eine Anfrage und eine zugehörige Antwort aussehen, ist es natürlich interessant, zu testen, wie viel Aufwand die tatsächliche Implementierung in PHP bedeutet. Zum Glück kann man dabei auf fertige (so gut das bei Open Source Projekten geht) Frameworks wie das von Dietrich Ayala [2] zurückgreifen. Weitere Implementierungen, auch in anderen Sprachen, finden sich bei SoapWare.org [3].

Unser Testbeispiel soll eine Webpage mit einem Formular zeigen, in dem man einen Nachnamen angeben kann. Mit diesem Namen als Parameter soll das Form-Script als SOAP Client dann den SOAP Server nach den gesamten Adressdaten zu diesem Namen fragen. Das Serverskript soll diese Daten in einer MySQL Tabelle der folgenden Struktur suchen:

Listing 1

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope
 xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/
                                   XMLSchema-instance"
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/
                                        soap/encoding/"
 xmlns:si="http://soapinterop.org/xsd"
 xmlns:ns6="http://soapinterop.org"
 SOAP-ENV:encodingStyle="http://schemas.
                         xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
 <ns6:getAdr>
  <query xsi:type="ns6:struct">
   <name xsi:type="xsd:string">huber</name>
  </query>
 </ns6:getAdr>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 2

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/
                                        soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/
                                   XMLSchema-instance"
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/
                                        soap/encoding/"
 xmlns:si="http://soapinterop.org/xsd"
 SOAP-ENV:encodingStyle="http://schemas.
                         xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
 <getAdrResponse>
  <getAdr>
   <nachname xsi:type="xsd:string">Huber</nachname>
   <vorname xsi:type="xsd:string">Gerhard</vorname>
   <strasse xsi:type="xsd:string">Benediktstr. 3</strasse>
   <plz xsi:type="xsd:string">50020</plz>
   <ort xsi:type="xsd:string">Koeln</ort>
   <tel xsi:type="xsd:string">0221/123456</tel>
  </getAdr>
 </getAdrResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 3

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://
                  schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
 <SOAP-ENV:Fault>
  <faultcode>SOAP-ENV:Server</faultcode>
  <faultstring>Server Error</faultstring>
  <detail>
   <e:myfaultdetails xmlns:e="Some-URI">
    <message>
     My application didn't work
    </message>
    <errorcode>
     1001
    </errorcode>
   </e:myfaultdetails>
  </detail>
 </SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 4

```
<?php

include("class.soap_client.php");
include("class.soap_server.php");

$server = new soap_server;

$server->add_to_map(
 "getAdr",
 array("struct"),
 array("struct")
);

function getAdr($qdata)
{
 $db=mysql_connect("localhost", "root", "v1");
 mysql_select_db("adressen");
 $query="select * from adressen where nachname
                          =".$qdata["name"]."';";
 $res=mysql_query($query);
 $dump=mysql_fetch_assoc($res);
 $data["nachname"]=$dump["nachname"];
 $data["vorname"]=$dump["vorname"];
 $data["strasse"]=$dump["strasse"];
 $data["plz"]=$dump["plz"];
 $data["ort"]=$dump["ort"];
 $data["tel"]=$dump["tel"];
 return $data;
}

$server->service($HTTP_RAW_POST_DATA);

?>
```

```
CREATE TABLE adressen (
 id int(11) NOT NULL default '0',
 vorname varchar(20) default NULL,
 nachname varchar(20) NOT NULL default '',
 strasse varchar(40) default NULL,
 plz varchar(10) default NULL,
 ort varchar(20) default NULL,
 tel varchar(20) default NULL,
 PRIMARY KEY (id),
 KEY vorname (vorname,nachname,tel),
 KEY id (id)
) TYPE=MyISAM;
```

Herr und Meister

Beginnen wir mit dem Serverskript *SOAP_SERVER.php* (Listing 4). Als Erstes müssen die Server und Client Klassen eingebunden und ein *soap_server* Objekt instanziiert werden:

```
include(„class.soap_client.php");
include(„class.soap_server.php");

$server = new soap_server;
```

Im Server muss auch die Client-Klasse inkludiert werden, da in dieser eine Reihe allgemeiner Definitionen zu finden sind. Dann bringt man dem Server bei, dass er nun eine neue Routine kennt:

```
$server->add_to_map(
 „getAdr",
 array(„struct"),
 array(„struct")
);
```

add_to_map wird mit drei Argumenten aufgerufen: dem Namen der einzubinden-

Listing 5

```
function getAdr($qdata)
{
 $db=mysql_connect("localhost", "user", "password");
 mysql_select_db("adressen");
 $query="select * from adressen where nachname
                                        ='".$qdata["name"]."';";
 $res=mysql_query($query);
 $dump=mysql_fetch_assoc($res);
 $data["nachname"]=$dump["nachname"];
 $data["vorname"]=$dump["vorname"];
 $data["strasse"]=$dump["strasse"];
 $data["plz"]=$dump["plz"];
 $data["ort"]=$dump["ort"];
 $data["tel"]=$dump["tel"];
 return $data;
}
```

Listing 6

```
<?

include("class.soap_client.php");

$nserver="paradiso"; // welchen Server konnektieren?
$method="getAdr";        // welche Methode aufrufen?

$method_params["getAdr"]["query"] = array("name"=> $name);

$servers[$nserver] = array(
                "soapaction" => "urn:soapinterop",
                "endpoint" => „http://$nserver:20080/SOAP_SERVER.php",
                "methodNamespace" => "http://soapinterop.org",
                "name" => $nserver);

print "<form action='$PHP_SELF' method='post'>";
?>
Name: <input name="name" type="text" size="30">       
<input name="__debug" type="checkbox" value="1">   Debugging Infos?
</form>
<hr>
<?
if($name) {
 $server = $servers[$nserver];
 $soap_message = new soapmsg($method,$method_params[$method],$server["methodNamespace"]);
 $soap = new soap_client($server["endpoint"]);
 if($return = $soap->send($soap_message,$server["soapaction"]))
 {
  // check for valid response
  if(get_class($return) == "soapval")
  {
   // fault?
   if(eregi("fault",$return->name))
    $status = "failed";
   else
    $status = "passed";
  }
  else
   print "Client could not decode server's response<br>";
 }
 else
  print "Was unable to send or receive.";
 if($__debug==1)
 {
  print "<strong>Request:</strong><br><xmp>$soap->outgoing_payload</xmp><br>";
  print "<strong>Response:</strong><br><xmp>$soap->incoming_payload</xmp>";
 }
 $data=$soap->response->decode();
 print "<table>";
 foreach($data["getAdr"] as $k => $v)
  print "<tr><td><b>$k</b></td><td>$v</td></tr>";
 print "</table>";
}

unset($soapmsg);
unset($soap);
?>
```

den Routine, einem Array mit den Typen der Input-Parameter und einem ebensolchen mit den Output-Parametern. Die Typenbezeichnungen entsprechen den XML Schema Datentypen (int, float, string, base64, struct u.a.).Diese Routine sollte man natürlich dann auch definieren (siehe Listing 5).

Als Letztes bleibt nur noch, den Server an die Arbeit zu schicken:

```
$server->service($HTTP_RAW_POST_DATA);
```

Die Client-Seite

Das Client Skript (Listing 6) beginnt mit dem obligatorischen Include der Clientklasse und der Definition des Servernamens und der Parameter für den SOAP Aufruf:

```
include(„class.soap_client.php“);

$nserver=“paradiso“;     // welchen Server konnektieren?
$method=“getAdr“;        // welche Methode aufrufen?

$method_params[$method][„query“] = array(„name“=>
$name);
```

Anschließend legt man in *$servers* die Verbindungsdaten zur Kontaktaufnahme mit dem SOAP Server an. Das Sammeln in einem Array ist vom API der SOAP Implementierung zwar nicht vorgeschrieben, erleichtert aber bei Skripten mit Kontakt zu verschiedenen SOAP Servern die Verwaltung der Daten.

Danach baut man ein ganz gewöhnliches Web Formular auf, welches im Prinzip nur ein Textfeld enthalten müsste. Damit man aber etwas von der dahinter liegenden Technik sieht, bauen wir eine Checkbox mit einem Debuggingflag ein, sodass neben dem üblichen Output auch der rohe Request und die Response ausgegeben werden.

Wie bei PHP-basierten Formularen üblich, überprüft man dann anhand einer Formularvariable, ob ein Submit erfolgt ist. Falls ja, folgen eine Reihe von Fehlertests und bei korrekter Antwort die Ausgabe der Ergebnisse.

Schauen wir uns zum Schluss die Serverparameter und Kontaktaufnahme zum SOAP Server etwas genauer an. Das *$servers Array* enthält eine Reihe von Angaben, die bei der Instanziierung der SOAP Message mittels der *soapmsg* Klasse benötigt werden:

- *endpoint* definiert als URL den Endpunkt der Aktion, also die URL, die für den HTTP Request verwendet werden soll. Daher enthält *endpoint* auch den Namen des PHP Skripts.
- *name* definiert einfach noch einmal den Servernamen, für den Fall, dass mehrere Server in *$servers* verwaltet werden.
- *methodNamespace* bestimmt, welche URL als Bezeichner für den Namespace der aufgerufenen Methode (in unserem Fall *ns6*) benutzt werden soll.
- *soapaction* ist ein von der W3C SOAP Recommendation vorgeschriebenes Feld für den HTTP Header, welches den beteiligten Servern, Proxies oder Firewalls deutlich machen soll, was der eintreffende Request genau für eine Funktion hat. Das Feld muss nicht gefüllt sein, wenn aus der Request URL alle nötigen Details des Requests hervorgehen. In diesem Fall darf das Feld leer sein. Im Header muss es trotzdem vorhanden sein.

Um den SOAP Server nun zu kontaktieren, erzeugt man eine SOAP Message mit den Parametern Methodenname, Parameter-Array und Methoden-Namespace:

```
$soap_message = new soapmsg($method,$method
        _params[$method],$server[„methodNamespace“]);
```

Dann instanziiert man einen neuen SOAP Client mit Angabe des Endpunktes der Aktion:

```
$soap = new soap_client($server[„endpoint“]);
```

Zum Schluss weist man den Client an, die Message mit einer bestimmten SOAP Action zu verschicken:

```
$return = $soap->send($soap_message,$server
                              [„soapaction“])
```

Der Returnwert der *send* Methode sollte ein Objekt der Klasse *soapval* sein. Falls dem so ist, kann man das im SOAP Server per *return* abgeschickte Array mit

```
$data=$soap->response->decode();
```

dekodieren. Das Ergebnis ist ein echtes PHP Array, wie ursprünglich von *getAdr* zurückgegeben. Die SOAP Implementierung führt das Kodieren und Dekodieren auch komplexer PHP-Returnwerte automatisch und völlig transparent durch.

Falls man die Low Level Kommunikation zwischen SOAP Server und Client sehen möchte, kann man den Inhalt der Variablen *outgoing_payload* und *incoming_payload* der Client-Klasse abfragen.

Ende gut, alles gut?

Wie wir gesehen haben, ist die Verwendung von SOAP in PHP leicht und gut verständlich. Die in diesem Beitrag verwandte SOAP Implementierung von Dietrich Ayala bietet über den reinen Remote Call hinaus, wie er hier vorgestellt wurde, auch eine WSDL Implementierung zur Beschreibung der implementierten SOAP Methoden.

Dem Einsatz von SOAP im PHP-Umfeld sind kaum Grenzen gesetzt. So ist es möglich, Java-basierte Storage Container oder Application Server anzubinden.

Auch existieren schon Business Applikationen wie Billing Systeme mit einer SOAP Schnittstelle, sodass der Zugriff auf eine Online-Rechnung oder die Änderung der eigenen Kundendaten mittels einer Webpage ohne direkten Eingriff in das Abrechnungssystem möglich werden.

Dr. Volker Göbbels ist Geschäftsführer der Arachnion GmbH & Co. KG. Die Arachnion (www.arachnion.de) befasst sich mit der Beratung und Implementierung im Bereich Web Applikationen und Billing Systeme sowie Schulungen im XML und PHP Umfeld. Dr. Göbbels ist Mitglied im Team ThinkPHP (www.thinkphp.de).

Links & Literatur

[1] www.w3.org/TR/SOAP/
[2] dietrich.ganx4.com/soapx4/
[3] www.soapware.org/