SOAP Overview

Tamas Szigyarto

Department of Computer Modelling and Multiple Processors Systems,
Faculty of Applied Mathematics and Control Processes,
St Petersburg State University,
Universitetskii prospekt 35,
Peterhof, St Petersburg 198504, Russia
szigyarto_tamas@inbox.ru

Abstract

This paper contains an overview of the new XML-based protocol called SOAP (***S***imple ***O***bject ***A***ccess ***P***rotocol). A key points of the SOA (***S***ervice ***O***riented ***A***rchitecrure) philosophy that needed to understand SOAP implementation and necessity considered. The basic idea of SOAP & its realization within Web Services architecture demonstrated. Also paper includes some comparisons SOAP with the related technologies like DCOM & CORBA.

To avoid some misunderstandings, the author should notice that this paper is naturally a compilation of the material that referenced in the last section. This paper purposes only one's object - to give to the students the conception of the SOAP.

Key words: XML, SOAP, service, SOA.

1 Introduction

Software As Service. The concept of "software as a service" is nothing new. In fact, it has existed almost as long as computing itself. Back in the 1950s, there was a lot of talk about how computing would ultimately become a utility like electricity and gas and that people would simply be able to plug into it.

However, despite, or possibly, because of the furious pace of innovation in computing, we still are largely at the stage where most large organizations spurn the grid and instead have their own private generation stations inside their networks. Organizations have spent millions of dollars with large software vendors to help them build their own home-grown, large, and complex IT "generating" station.

There have been many initiatives over the years to deliver an SOA. We can look at DCE, CORBA, and DCOM as some recent examples. However, the emergence of the Internet and a set of commonly agreed software standards has presented an opportunity to overcome the shortcomings of previous attempts. The building blocks are in place to realize the vision of "software as a service".

Starting with the service. Let consider the next example. If you are a telecommunications provider, the services you provide to your customers are built around provisioning, billing, and customer care. People call you up and ask you to activate a telephone line, they then make calls on that telephone line, you send them a bill for those calls, and they call you when they are having problems with the service or the bill.

There is a lot of technology involved in providing those services to the customer. Many different systems need to be connected together to enable the service - historically those services grew from the technology outwards. That means that the way a service was created and delivered to the customer was dictated by the available technology.

The whole goal of SOA is to try and reverse that trend. The premise of SOA is to start with the service (as it is defined by the people who want to provide it) and then work backwards into the technology. It is, as it says, a Service-Oriented Architecture. This is a fundamental and profound change in the way we think about information systems and is also the main reason why SOA might succeed this time around. If you look at any of the previous attempts at a SOA, you can see that they were very technology-centric. To demonstrate this point, let me use a quote from IBM:

"So it (SOA) basically boils down to distributed computing with standards that tell us how to invoke different applications as services in a secure and reliable way and then how we can link the different services together using choreography to create business processes. And then finally so that we can manage these services so that ultimately we can manage and monitor our business performance."

While this is technologically valid, it is missing the point of SOA. Again, we're focused on the technology that enables SOA and not on SOA itself. This is one of the biggest hurdles in making SOA work.

Now we can turn to consider the SOAP & the abilities that it gives to us to realize the SOA architecture in natural way.

2 What Is SOAP?

SOAP how it is. If you had asked anyone what SOAP meant several years ago, they would have probably said something like "it's for making DCOM and CORBA (e.g., Remote Procedure Calls (RPC)) work over the Internet". The original authors admit they were focused on "accessing objects" back then, but over time it became desirable for SOAP to serve a much broader audience. Hence, the focus of the specification quickly moved away from objects towards a generalized XML messaging framework.

SOAP is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. SOAP uses XML technologies to define an extensible messaging framework, which provides a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics.

This definition really gets to the heart of what SOAP is about today. SOAP defines a way to move XML messages from point A to point B (see Figure 1). It does this by providing an XML-based messaging framework that is 1) extensible, 2) usable over a variety of underlying networking protocols, and 3) independent of programming models. Let's discuss each of these three characteristics in a bit more detail.

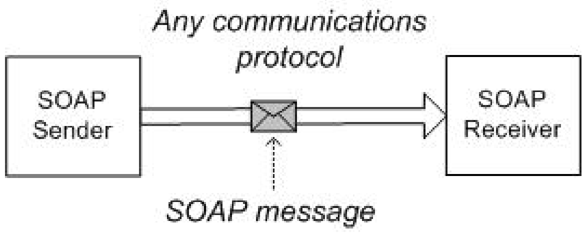


Figure 1. Simple SOAP messaging

First, SOAP extensibility is key. When the acronym stood for something, "S" meant "Simple". Simplicity remains one of SOAP's primary design goals as evidenced by SOAP's lack of various distributed system features such as security, routing, and reliability to name a few. SOAP defines a communication framework that allows for such features to be added down the road as layered extensions. Microsoft, IBM, and other software vendors are actively working on a common suite of SOAP extensions that will add many of these features that most developers expect.

Second, SOAP can be used over any transport protocol such as TCP, HTTP, SMTP, or even MSMQ (Microsoft Message Queue)(see Figure 1). In order to maintain interoperability, however, standard protocol bindings need to be defined that outline the rules for each environment. The SOAP specification provides a flexible framework for defining arbitrary protocol bindings and provides an explicit binding today for HTTP since it's so widely used.

Third, SOAP allows for any programming model and is not tied to RPC. Most developers immediately equate SOAP to making RPC calls on distributed objects (since it was originally about "accessing objects") when in fact, the fundamental SOAP model is more akin to traditional messaging systems like MSMQ. SOAP defines a model for processing individual, one-way messages. You can combine multiple messages into an overall message exchange. Figure 1 illustrates a simple one-way message where the sender doesn't receive a response. The receiver could, however, send a response back to the sender (see Figure 2). SOAP allows for any number of message exchange patterns (MEPs), of which request/response is just one. Other examples include solicit/response (the reverse of request/response), notifications, and long running peer-to-peer conversations.

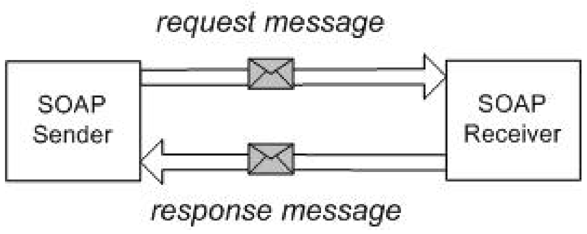


Figure 2. Request/response message exchange pattern

Developers often confuse request/response with RPC when they're actually quite different. RPC uses request/response, but request/response isn't necessarily RPC. RPC is a programming model that allows developers to work with method calls. RPC requires a translation of the method signature into SOAP messsages. Due to the popularity of RPC, SOAP outlines a convention for using RPC with SOAP.

Armed with these three major characteristics, the SOAP messaging framework facilitates exchanging XML messages in heterogeneous environments where interoperability has long been a challenge.

Messaging Framework. The core section of the SOAP specification is the messaging framework. The SOAP messaging framework defines a suite of XML elements for "packaging" arbitrary XML messages for transport between systems.

The framework consists of the following core XML elements: Envelope, Header, Body, and Fault, all of which are from the http://schemas.xmlsoap.org/soap/envelope/namespace in SOAP 1.1. Now we can take a look at XML Schema definition for SOAP 1.1 in the following code.

SOAP 1.1 XML Schema Definition

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://schemas.xmlsoap.org/soap/envelope/"
  targetNamespace="http://schemas.xmlsoap.org/soap/envelope/">

  <!-- Envelope, header and body -->
  <xs:element name="Envelope" type="tns:Envelope" />
  <xs:complexType name="Envelope" >
    <xs:sequence>
      <xs:element ref="tns:Header" minOccurs="0" />
      <xs:element ref="tns:Body" minOccurs="1" />
      <xs:any namespace="##other" minOccurs="0"
       maxOccurs="unbounded" processContents="lax" />
    </xs:sequence>
    <xs:anyAttribute namespace="##other"
     processContents="lax" />
  </xs:complexType>

  <xs:element name="Header" type="tns:Header" />
  <xs:complexType name="Header" >
    <xs:sequence>
      <xs:any namespace="##other" minOccurs="0"
       maxOccurs="unbounded" processContents="lax" />
    </xs:sequence>
    <xs:anyAttribute namespace="##other"
     processContents="lax" />
  </xs:complexType>
```

```
<xs:element name="Body" type="tns:Body" />
  <xs:complexType name="Body" >
    <xs:sequence>
      <xs:any namespace="##any" minOccurs="0"
       maxOccurs="unbounded" processContents="lax"
/>
    </xs:sequence>
    <xs:anyAttribute namespace="##any"
     processContents="lax" />
  </xs:complexType>


  <!-- Global Attributes -->
  <xs:attribute name="mustUnderstand" default="0" >

     <xs:simpleType>
     <xs:restriction base='xs:boolean'>
      <xs:pattern value='0|1' />
    </xs:restriction>
   </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="actor" type="xs:anyURI" />

  <xs:simpleType name="encodingStyle" >
    <xs:list itemType="xs:anyURI" />
  </xs:simpleType>
```

```
<xs:attribute name="encodingStyle"
  type="tns:encodingStyle" />
 <xs:attributeGroup name="encodingStyle" >
   <xs:attribute ref="tns:encodingStyle" />
 </xs:attributeGroup>

 <xs:element name="Fault" type="tns:Fault" />
 <xs:complexType name="Fault" final="extension" >
   <xs:sequence>
     <xs:element name="faultcode" type="xs:QName" />
     <xs:element name="faultstring" type="xs:string" />
     <xs:element name="faultactor" type="xs:anyURI"
     minOccurs="0" />
     <xs:element name="detail" type="tns:detail"
     minOccurs="0" />
   </xs:sequence>
 </xs:complexType>

 <xs:complexType name="detail">
   <xs:sequence>
     <xs:any namespace="##any" minOccurs="0"
     maxOccurs="unbounded" processContents="lax" />
   </xs:sequence>
   <xs:anyAttribute namespace="##any"
    processContents="lax" />
 </xs:complexType>

</xs:schema>
```

If you check out the complexType definition for Envelope, you can quickly learn how these elements relate to each other. The following message template illustrates the structure of a SOAP Envelope:

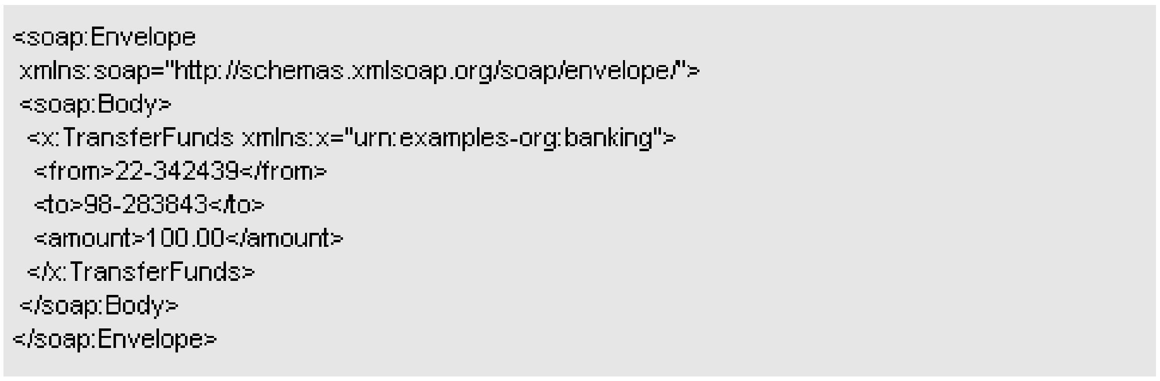
```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Header> <!-- optional -->
   <!-- header blocks go here... -->
 </soap:Header>
 <soap:Body>
   <!-- payload or Fault element goes here... -->
 </soap:Body>
</soap:Envelope>
```

The Envelope element is always the root element of a SOAP message. This makes it easy for applications to identify "SOAP messages" by simply looking at the name of the root element. Applications can also determine the version of SOAP being used by inspecting the Envelope element's namespace name.

The Envelope element contains an optional Header element followed by a mandatory Body element. The Body element represents the message payload. The Body element is a generic container in that it can contain any number of elements from any namespace.

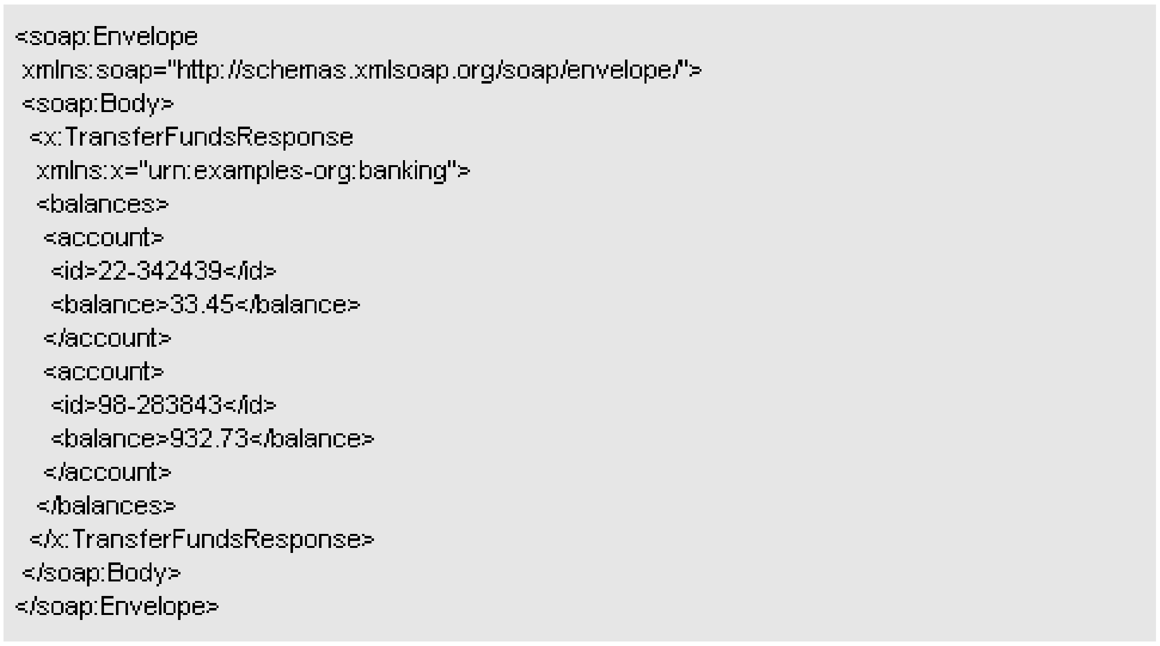
This is ultimately where the data goes that you're trying to send.

For example, the following SOAP message represents a request to transfer funds between bank accounts:



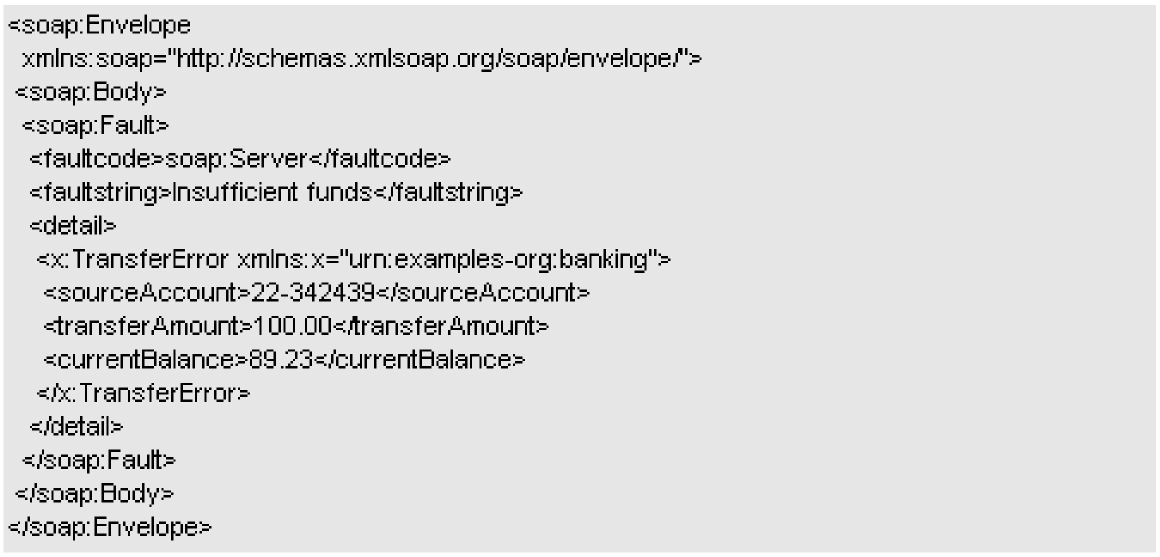
```
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <x:TransferFunds xmlns:x="urn:examples-org:banking">
   <from>22-342439</from>
   <to>98-283843</to>
   <amount>100.00</amount>
  </x:TransferFunds>
 </soap:Body>
</soap:Envelope>
```

If the receiver supports request/response and it is able to process the message successfully, it would send another SOAP message back to the initial sender. In this case, the response information would also be contained in the Body element as illustrated in this example:



```
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <x:TransferFundsResponse
   xmlns:x="urn:examples-org:banking">
   <balances>
    <account>
     <id>22-342439</id>
     <balance>33.45</balance>
    </account>
    <account>
     <id>98-283843</id>
     <balance>932.73</balance>
    </account>
   </balances>
  </x:TransferFundsResponse>
 </soap:Body>
</soap:Envelope>
```

The messaging framework also defines an element named Fault for representing errors within the Body element when things go wrong. This is essential because without a standard error representation, every application would have to invent their own making it impossible for generic infrastructure to distinguish between success and failure. The following sample SOAP message contains a Fault element that indicates an ”Insufficient Funds” error occurred while processing the request:



```
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <soap:Fault>
   <faultcode>soap:Server</faultcode>
   <faultstring>Insufficient funds</faultstring>
   <detail>
    <x:TransferError xmlns:x="urn:examples-org:banking">
     <sourceAccount>22-342439</sourceAccount>
     <transferAmount>100.00</transferAmount>
     <currentBalance>89.23</currentBalance>
    </x:TransferError>
   </detail>
  </soap:Fault>
 </soap:Body>
</soap:Envelope>
```

The Fault element must contain a faultcode followed by a faultstring element. The faultcode element classifies the error using a namespace-qualified name, while the faultstring element provides a human readable explanation of the error (similar to how HTTP works). Table 1 provides brief descriptions of the SOAP 1.1 defined fault codes (all of which are in the http://schemas.xmlsoap.org/soap/envelope/namespace).

The Fault element may also contain a detail element for providing details about the error, which may help clients diagnose the problem, especially in the case of Client and Server fault codes.

Table 1. SOAP 1.1 Fault Codes

Name	**Meaning**
VersionMismatch	The processing party found an invalid namespace for the SOAP Envelope element.
MustUnderstand	An immediate child element of the SOAP Header element that was either not understood or not obeyed by the processing party contained a SOAP mustUnderstand attribute with a value of ”1”.
Client	The Client class of errors indicates that the message was incorrectly formed or did not contain the appropriate information in order to succeed. It is generally an indication that the message should not be resent without change.
Server	The Server class of errors indicates that the message could not be processed for reasons not directly attributable to the contents of the message, but rather to the processing of the message. For example, processing could include communicating with an upstream processor, which didn’t respond. The message may succeed if re-sent at a later point in time.

Now imagine that you want to add some authentication information to the original message so the receiver can determine whether the sender has sufficient rights to execute the transfer. A way to do this would be to add the credentials information into the body as shown here:

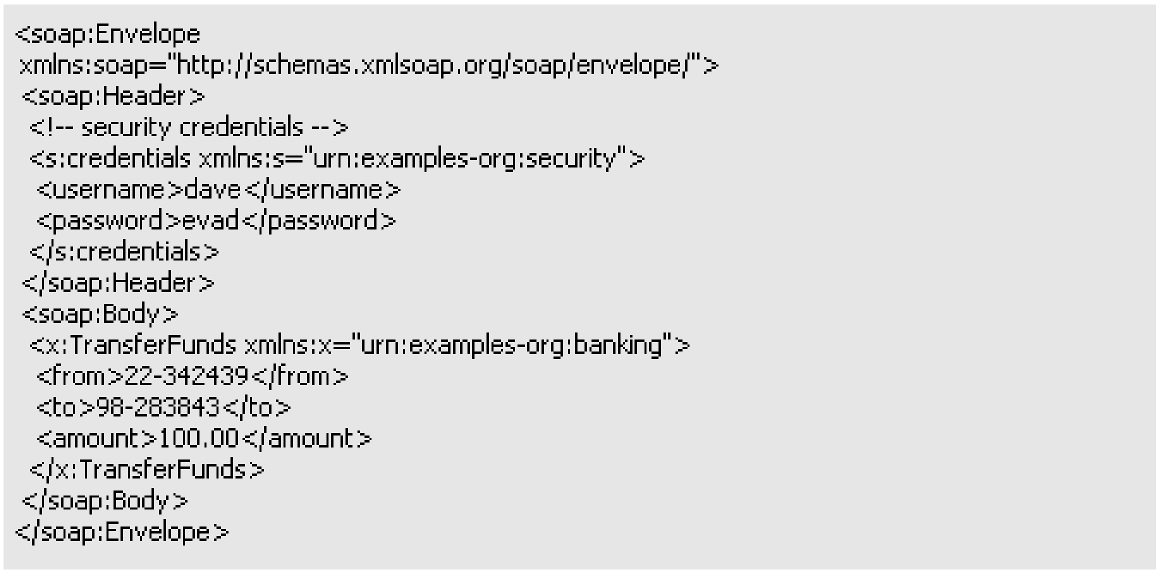
```
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
   <x:TransferFunds xmlns:x="urn:examples-org:banking">
    <from>22-342439</from>
    <to>98-283843</to>
    <amount>100.00</amount>
    <!-- security credentials -->
    <credentials>
     <username>dave</username>
     <password>evad</password>
    </credentials>
   </x:TransferFunds>
  </soap:Body>
</soap:Envelope>
```

Going down this path requires every operation that needs authentication to deal with the credentials. It also means that other applications in need of security must develop their own solutions to the problem; ultimately, interoperability suffers. For common needs such as security, it makes more sense to define standard SOAP headers that everyone agrees on. Then, vendors can build support for the extended functionality into their generic SOAP infrastructure and everyone wins. This approach increases developer productivity and helps ensure higher levels of interoperability at the same time. This is exactly the type of thing the SOAP extensibility model was designed to facilitate.

Extensibility. Most existing protocols make a distinction between control information (e.g., headers) and message payload. SOAP is no different in this regard. The SOAP Header and Body elements provide the same distinction in the easy-to-process world of XML. Besides ease of use, the key benefit of the extensible Envelope is that it can be used with any communications protocol.

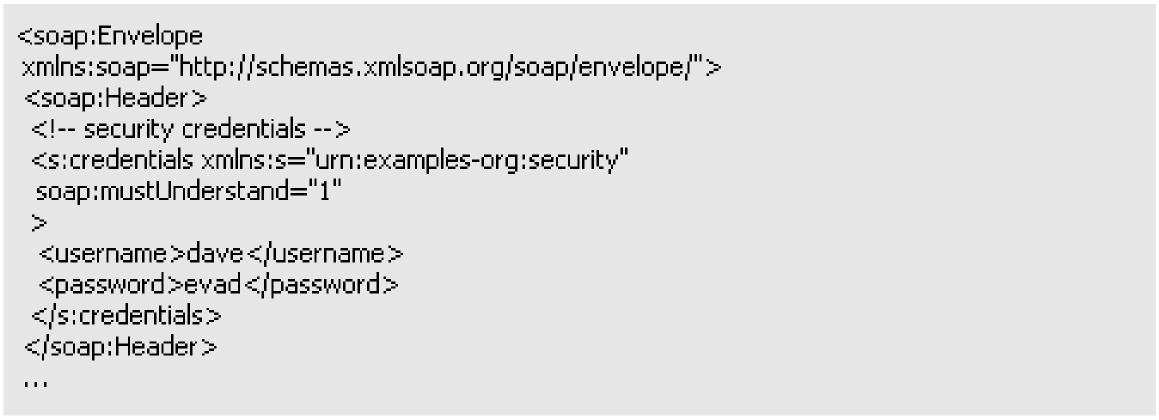
Headers have always played an important role in application protocols, like HTTP, SMTP, etc., because they allow the applications on both ends of the wire to negotiate the behavior of the supported commands. Although the SOAP specification itself doesn't define any built-in headers, headers will eventually play an equally important role in SOAP. As GXA (Global XML Web Services Architecture) matures and SOAP headers become standardized, it will become easier for developers to define rich application protocols, without having to reinvent the wheel each time.

The Header element, like the Body element, is a generic container for control information. It may contain any number of elements from any namespace (other than the SOAP namespace). Elements placed in the Header element are referred to as header blocks. As with other protocols, header blocks should contain information that influences payload processing. Hence, this is the right place to put something like a credentials element that helps control access to the operation:



```
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Header>
  <!-- security credentials -->
  <s:credentials xmlns:s="urn:examples-org:security">
   <username>dave</username>
   <password>evad</password>
  </s:credentials>
 </soap:Header>
 <soap:Body>
  <x:TransferFunds xmlns:x="urn:examples-org:banking">
   <from>22-342439</from>
   <to>98-283843</to>
   <amount>100.00</amount>
  </x:TransferFunds>
 </soap:Body>
</soap:Envelope>
```

Header blocks can also be annotated with a global SOAP attribute named mustUnderstand to indicate whether or not the receiver is required to understand the header before processing the message. The following example illustrates how to require the processing of the credentials header:



```
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Header>
  <!-- security credentials -->
  <s:credentials xmlns:s="urn:examples-org:security"
   soap:mustUnderstand="1"
  >
   <username>dave</username>
   <password>evad</password>
  </s:credentials>
 </soap:Header>
 ...
```

If a header block is annotated with mustUnderstand="1" and the receiver wasn't designed to support the given header, the message shouldn't be processed and a Fault should be returned to the sender (with a soap:MustUnderstand status code). When mustUnderstand="0" or the mustUnderstand attribute isn't present, the receiver can ignore those headers and continue processing. The mustUnderstand attribute plays a central role in the overall SOAP processing model.

Processing Model. SOAP defines a processing model that outlines rules for processing a SOAP message as it travels from a SOAP sender to a SOAP receiver. Figure 1 illustrates the simplest SOAP messaging scenario, where there's one application (SOAP sender) sending a SOAP message to another application (SOAP receiver).

The processing model, however, allows for more interesting architectures like the one in Figure 3, which contains multiple intermediary nodes. Further, we will use the term SOAP node to refer to any application that processes SOAP messages, whether it's the initial sender, an intermediary, or the ultimate receiver.

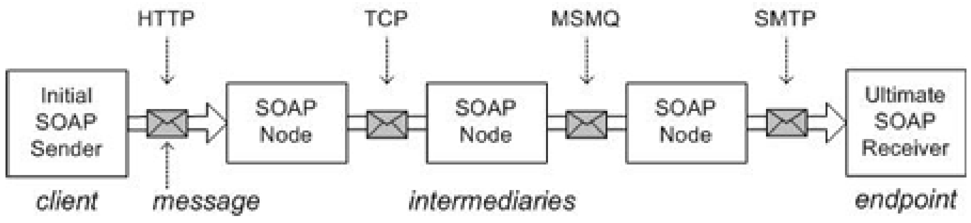


Figure 3. Sophisticated SOAP messaging

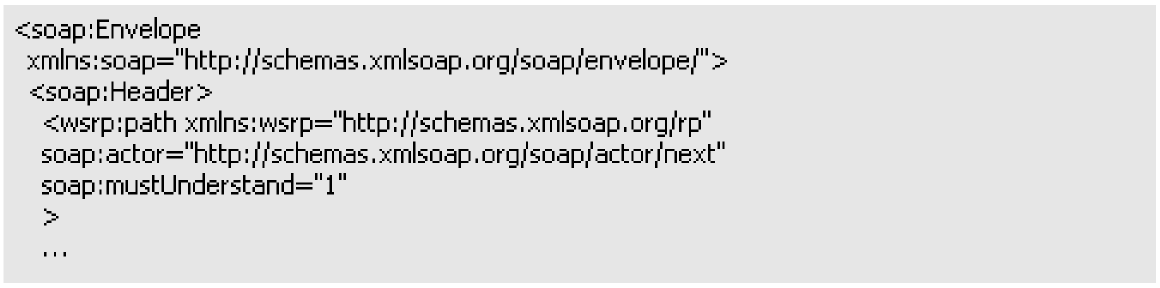
An intermediary sits between the initial sender and the ultimate receiver and intercepts SOAP messages. An intermediary acts as both a SOAP sender and a SOAP receiver at the same time. Intermediary nodes make it possible to design some interesting and flexible networking architectures that can be influenced by message content. SOAP routing is a good example of something that heavily leverages SOAP intermediaries.

While processing a message, a SOAP node assumes one or more roles that influence how SOAP headers are processed. Roles are given unique names (in the form of URIs) so they can be identified during processing. When a SOAP node receives a message for processing, it must first determine what roles it will assume. It may inspect the SOAP message to help make this determination.

Once it determines the roles in which it will act, the SOAP node must then process all mandatory headers (marked mustUnderstand="1") targeted at one of its roles. The SOAP node may also choose to process any optional headers (marked mustUnderstand="0") targeted at one of its roles.

SOAP 1.1 only defines a single role named http://schemas.xmlsoap.org/soap/actor/next (next, for short). Every SOAP node is required to assume the next role. Hence, when a SOAP message arrives at any given SOAP node, the node must process all mandatory headers targeted at the next role, and it may choose to process optional headers also targeted at the next role. In addition to next, SOAP 1.2 defines a few more roles (see Table 2) and applications are allowed to define custom roles as well.

SOAP headers target specific roles through the global actor attribute (the attribute is named role in SOAP 1.2). If the actor attribute isn't present, the header is targeted at the ultimate receiver by default. The following SOAP message illustrates how to use actor:



```
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Header>
  <wsrp:path xmlns:wsrp="http://schemas.xmlsoap.org/rp"
  soap:actor="http://schemas.xmlsoap.org/soap/actor/next"
  soap:mustUnderstand="1"
  >
  ...
```

Since the wsrp:path header is targeted at the next role and marked as mandatory (mustUnderstand="1"), the first SOAP node to receive this message is required to process it according to the header block's specification, in this case WS-Routing. If the SOAP node wasn't designed to understand a mandatory header targeted at one of its role, it is required to generate a SOAP fault, with a soap:MustUnderstand status code, and discontinue processing. The SOAP Fault element provides the faultactor child element to specify who caused the fault to happen within the message path. The value of the faultactor attribute is a URI that identifies the SOAP node that caused the fault.

If a SOAP node successfully processes a header, it's required to remove the header from the message. SOAP nodes are allowed to reinsert headers, but doing so changes the contract partiesit's now between the current node and the next node the header targets. If the SOAP node happens to be the ultimate receiver, it must also process the SOAP body.

Table 2. SOAP 1.2 Roles

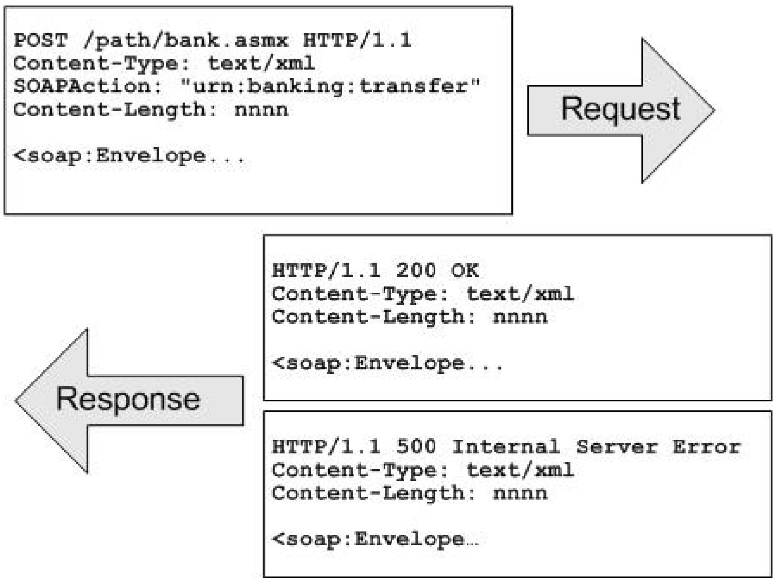
SOAP Role Name	**Description**
http://www.w3.org/2002/06/soap-envelope/role/next	Each SOAP intermediary and the ultimate SOAP receiver MUST act in this role and MAY additionally assume zero or more other SOAP roles.
http://www.w3.org/2002/06/soap-envelope/role/none	SOAP nodes MUST NOT act in this role.
http://www.w3.org/2002/06/soap-envelope/role/ultimateReceiver	To establish itself as an ultimate SOAP receiver, a SOAP node MUST act in this role. SOAP intermediaries MUST NOT act in this role.

Protocol Bindings. An interesting aspect of Figure 3 is that SOAP enables message exchange over a variety of underlying protocols. Since the SOAP messaging framework is independent of the underlying protocol, each intermediary could choose to use a different communications protocol without affecting the SOAP message. Standard protocol bindings are necessary, however, to ensure high levels of interoperability across SOAP applications and infrastructure.

A concrete protocol binding defines exactly how SOAP messages should be transmitted with a given protocol. In other words, it defines the details of how SOAP fits within the scope of another protocol, which probably has a messaging framework of its own along with a variety of headers. What the protocol binding actually defines depends a great deal on the protocol's capabilities and options. For example, a protocol binding for TCP would look much different than one for MSMQ or another for SMTP.

The SOAP 1.1 specification only codifies a protocol binding for HTTP, due to its wide use. SOAP has been used with protocols other than HTTP but the implementations weren't following a standardized binding. There's nothing wrong with forging ahead without standard protocol bindings in place as long as you're prepared to deal with interoperability issues once you try to integrate with other SOAP implementations using the same protocol.

The HTTP protocol binding defines the rules for using SOAP over HTTP. SOAP request/response maps naturally to the HTTP request/response model. Figure 4 illustrates many of the SOAP HTTP binding details.



```
POST /path/bank.asmx HTTP/1.1
Content-Type: text/xml
SOAPAction: "urn:banking:transfer"
Content-Length: nnnn

<soap:Envelope...
```

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: nnnn

<soap:Envelope...
```

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml
Content-Length: nnnn

<soap:Envelope...
```

Figure 4. SOAP HTTP binding

The Content-Type header for both HTTP request and response messages must be set to text/xml (application/soap+xml in SOAP 1.2). As for the request message, it must use POST for the verb and the URI should identify the SOAP processor. The SOAP specification also defines a new HTTP header called SOAPAction, which must be present in all SOAP HTTP requests (even if empty). The SOAPAction header is meant to express the intent of the message. As for the HTTP response, it should use a 200 status code if no errors occurred or 500 if the body contains a SOAP Fault.

3 Why SOAP?

Industry Acceptance for SOAP. SOAP has attracted great attention in the software development community over the past year. SOAP has been implemented as the standards have evolved. This means that developers have been able to download and use the technology, rather than just read specifications and industry analysis.

SOAP has been described as a disruptive technology because it changes both the way software will be developed and the way industry rivals are cooperating. The two companies that showed early leadership in Web Services, Microsoft and IBM, have clearly demonstrated that they see real value in rapid and widespread acceptance of the Web Services paradigm.

The association of Microsofts .NET strategy with the more general industry effort in Web Services is proving highly credible in the corporate marketplace. This is in contrast to Microsofts previous effort at developing an Internet platform, Windows DNA, which arrived late and lost out to J2EE in the corporate market. A key benefit for IBM is that Web Services provide a link between its various generations of technology.

Many of the key industry leaders have announced their support for SOAP, including HP, SAP, Software AG, Sun Microsystems, and Oracle. One of the most interesting developments since the introduction of SOAP has been the emergence of Web Services Description Language (WSDL) and Universal Description, Discover, and Integration (UDDI) to provide a unified Web Services model for the future of distributed computing over the Internet. This in turn has lead to the release of Web Services platforms, with SOAP servers being the core technology in the initial releases.

An interesting aspect of SOAP is that it is not technologically advanced being based on remote procedure calls and XML and initially using HTTP as the transport layer. Using well understood and accepted technology has ensured that there have been relatively few technology disputes.

SOAP Client Implementations and Interoperability. A very significant number of SOAP clients (typically API's that generate the SOAP payload) have been developed. In theory, it is possible to implement SOAP clients in any programming language on any operating system. This is one of the benefits of the ***"S"*** in the SOAP acronym.

Some of the early SOAP client implementations suffered from interoperability issues (incomplete standards compliance) or usability issues (for example, Apache SOAP requires hand-coded serialization). Apache SOAP is one of the most widely used SOAP implementations for learning about Web Services and non-commercial deployments. However, Apache SOAP requires type information to be passed with the SOAP message (not part of the SOAP 1.1 specification). One technology that can help with interoperability issues with the many SOAP client implementations is XSLT. This can be used to map the SOAP message generated by the SOAP client with the message expected by the SOAP server.

SOAP Transport. Most SOAP servers currently use HTTP as the transport layer for the XML payload in SOAP messages. HTTP satisfies a number of requirements of an Internet transport:

- Ubiquity
- Firewall - friendliness
- Simplicity
- Statelessness (makes for graceful failure)
- Scalability (Web servers are proven to support extremely high traffic)
- Readily capable of being made secure

However, the SOAP 1.1 specification clearly states that the transport layer is flexible. There are a number of SOAP implementations that support other transport layers, such as:

- HTTPS - using SSL provides security
- SMTP - enables asynchronous SOAP requests SOAP Report
- JMS (Java Message Service) - provides the benefit of tight integration with the J2EE platform

It can be expected that other transports, such as MQSeries or FTP, will be supported eventually. IBM has an interesting proposal for HTTP-R. This is to provide a reliable transport layer for SOAP messages.

SOAP Performance. One of the early objections to the use of SOAP has been performance, especially when used as an alternative to CORBA and DCOM. However, the more recent versions of SOAP servers have only slightly lower performance than RMI (Remote Method Invocation). This is mainly due to two factors: the use of SAX (Simple API for XML) rather than DOM (Document Object Model) to parse the messages and optimization lessons learned from earlier product releases. The use of SAX parsers has increased throughput, reduced memory overhead, and improved scalability.

Secure SOAP. It is commonly stated that SOAP is not secure. This is partly due to the widely discussed fact that SOAP over HTTP passes through firewalls. However, the SOAP 1.1 specification makes it clear that SOAP can be implemented over any suitable transport protocol. An obvious example is HTTPS (although SOAP over SMTP is useful for asynchronous messaging).

There are other standard security features that can be implemented immediately with SOAP-based Web Services, such as authentication and authorization. These are well-known and well-understood security models that are the best way to implement security for Web Services. Other, XML-based security standards are under development, but are currently unproven and require new skill sets that may not be available.

SOAP and EAI. There has been considerable speculation that current Enterprise Application Integration (EAI) products will be made redundant by SOAP. However, while they may lose market share in some smaller projects, there is still a role for heavy-duty, enterprise-grade EAI products that integrate with more obscure legacy systems or provide unusually high qualities of service. What is more significant is that SOAP will encourage and enable a whole new generation of EAI projects that were not previously possible due to technical and cost constraints. But there will still be room on the market for other integration products.

Criteria for Assessing SOAP Servers. There are already several commercial SOAP servers and Web Services platforms available. Some of these also provide support for WSDL and UDDI. When assessing the viability of SOAP servers, the key issues to address include:

- Compliance with the standards
- Interoperability testing and XSLT support
- Support for Microsoft SOAP
- Support for Apache SOAP
- Security features
- Support for multiple transport layers
- Performance

4 Web Services Solution

In the previous section we mentioned some words about Web Services technology and about place of SOAP within it. Now I want to consider this in more detailed way to give some idea about how we can use flexibility of the SOAP on a practice.

What is the Web Service? With Web Services, information sources become components that you can use, re-use, mix, and match to enhance Internet and intranet applications ranging from a simple currency converter, stock quotes, or dictionary to an integrated, portalbased travel planner, procurement workflow system, or consolidated purchase processes across multiple sites. Each is built as stack of layers, or a narrative format. Each vendor, standards organization, or marketing research firm defines Web Services in a slightly different way. Gartner, for instance, defines Web Services as "loosely coupled software components that interact with one another dynamically via standard Internet technologies." Forrester Research takes a more open approach to Web Services as "automated connections between people, systems and applications that expose elements of business functionality as a software service and create new business value." For these reasons, the architecture of a Web Services stack varies from one organization to another.

The number and complexity of layers for the stack depend on the organization. Each stack requires Web Services interfaces to get a Web Services client to speak to an Application Server, or Middleware component, such as CORBA, J2EE, or .NET.

Web Services, at a basic level, can be considered as a universal client/server architecture that allows disparate systems to communicate with each other without using proprietary client libraries. We can points out that this architecture simplifies the development process typically associated with client/server applications by effectively eliminating code dependencies between client and server and the server interface information is disclosed to the client via a configuration file encoded in a standard format (e.g. WSDL). Doing so allows the server to publish a single file for all target client platforms.

Web Services stack of layers. Here we show the basic Web Services layers that are taking from WebServices.Org.

Table 3. Web Services Stack.

Layer	**Example**
Workflow, discovery & registries	UDDI
Service description language	WSDL
Messaging	SOAP \ XML protocol
Transport protocols	HTTP, HTTPS, SMTP

Workflow, Discovery, Registries. Web Services that can be exposed may, for example, get information on credit validation activities from a public directory or registry, such as Universal Description, Discovery and Integration (UDDI). The ebXML, E-Services Village, BizTalk.org, and xml.org registries and Bowstreet's (a stock service brokerage) Java-based UDDI (jUDDI) are other directories that could be used with UDDI in conjunction with Web Services for business-to-business (B2B) transactions in a complex EAI infrastructure under certain conditions. Web Services is still primarily an interfacing architecture, and needs an integration platform to which it is connected. Such an integration platform would cover the issue of integrating an installed base of applications that cannot work as Web Services yet.

The first release of UDDI's Business Registry became fully operational in May 2001, enabling businesses to register and discover Web Services via the Internet. Its original intent was to enable electronic catalogues in which businesses and services could be listed. The UDDI specification defines a way to publish and discover information about services. In November 2001, the UDDI Business Registry v2 beta became publicly available.

Hewlett Packard Company, IBM, Microsoft, and SAP launched beta implementation of their UDDI sites that have conformed to the latest specification, including enhanced support for deploying public and private Web Service registries, and the interface (SOAP/HTTP

API) that the client could use to interact with the registry server. In addition to the public UDDI Business Registry sites, enterprises can also deploy private registries on their intranet to manage internal Web Services using the UDDI specification. Access to internal Web Service information may also be extended to a private network of business partners.

Service Description Language. As you move further down the stack, you need WSDL to connect to a Web Service. This language is an XML format for describing network services. With it, service requesters can search for and find the information on services via UDDI, which, in turn, returns the WSDL reference that can be used to bind to the service. Web Service Conversational Language (WSCL) helps developers use the XML Schema to better describe the structure of data in a common format (say, with new data types) the customers, Web browsers, or indeed any XML enabled software programs can recognize. This protocol can be used to specify a Web Service interface and to describe service interactions.

Messaging. Now, we get to the Messaging layer in the stack where SOAP acts as the envelope for XML-based messages, covering message packaging, routing, guaranteed delivery and security. Messages are sent back and forth regarding the status of various Web Services as the work progresses (say, from customer order to shipping product out of the warehouse).

Transport Protocols. When a series of messages completes its rounds, the stack goes to its last layer: the transport layer using Hypertext Transfer Protocol (HTTP), Secure HTTP (HTTPS) or Standard Mail Transfer Protocol (SMTP). Then, each Web Service takes a ride over the Internet to provide a service requester with services or give a status report to a service provider or broker.

5 Conclusion

Now we stand on a way to review all the paper says. So, as we considered, today developers have a very flexible (by its nature) tool such as SOAP &, as its implementation, Web Services technology, by which they can create a distributed systems that interact through the Internet. It's necessary to mark, that those technology less complex & more flexible (in terms of integration) then previous such CORBA, DCOM e.t.c.

The widespread acceptance of SOAP means that almost all platforms and applications can be expected to eventually provide SOAP interfaces. This means that Web Service platforms, with SOAP servers at their core, will need to evolve from their current role as providing adaptors to expose existing logic (by generating WSDL representations of the back-end logic). It has been suggested that the future of SOAP servers is as Web Services Platforms that will be a "design center" in their own rightwhere new Web Services are developed, hosted, and made available through UDDI.

References

[1] **Clear Thinking** Series by Annrai O'Toole 2004, *Cape Clear Software Documentation*, http://www.capeclear.com/clear_thinking/

[2] w3c recomendations & notes, http://www.w3c.org/TR/SOAP/

[3] Understanding SOAP, Aaron Skonnard, 2003 *MSDN Library*, http://msdn.microsoft.com/

[4] An XML Overview Towards Understanding SOAP, Scott Seely, 2001 *MSDN Library*, http://msdn.microsoft.com/

[5] Web Services Architectures, Judith M. Myerson, http://www.webservicesarchitect.com/