INTERNET-DRAFT

D. Box
DevelopMentor

G. Kakivaya
A. Layman
S. Thatte
Microsoft
Corporation

D. Winer
Userland Software

Document: <draft-box-http-soap-01.txt>
Category: Informational

November 1999

SOAP: Simple Object Access Protocol

Status of this Memo

1. Abstract

SOAP defines an RPC mechanism using XML for client-server interaction across a network by using the following mechanisms:

* HTTP as the base transport
* XML documents for encoding of invocation requests and responses

2. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in

this document are to be interpreted as described in RFC-2119 [11].

The namespace prefix "SOAP" is used in this document to represent whatever prefix actually appears in the XML instance and is associated with the SOAP namespace URI "urn:schemas-xmlsoap-org:soap.v1". The prefix "xsd" represents whatever prefix is associated with the XML Schemas Specification URI [12].

Namespace URIs such of the general form "some-URI" represent some application-dependent or context-dependent URI.

3. Introduction

SOAP defines an "XML-RPC" protocol for client-server interaction across a network by using the following mechanisms:

* HTTP as the base transport
* XML documents for encoding of invocation requests and responses

SOAP is both low-entry and high-function, capable of use for simple stateless remote procedure calls as well as rich object systems.

SOAP works with today's deployed World Wide Web and provides extensibility mechanisms for future enhancements. For example, SOAP supports submitting invocations using both POST and M-POST.

3.1. Goals

* Provide a standard object invocation protocol built on Internet standards, using HTTP as the transport and XML for data encoding.
* Create an extensible protocol and payload format that can evolve.

3.2. Non-Goals

Define all aspects of a distributed object system, including the following:

* Distributed garbage collection
* Bi-directional HTTP communications
* Boxcarring or pipelining of messages
* Objects-by-reference (which requires distributed garbage collection and bi-directional HTTP)
* Activation (which requires objects-by-reference)

3.3. Examples of SOAP Calls

The call is to a StockQuote server, and the method is

GetLastTradePrice. The method takes one string parameter, ticker, and returns a float.

It uses the SOAP namespace to disambiguate SOAP keywords in the payload and a method namespace to disambiguate method keywords in the payload.

The root Envelope element tag name is used to disambiguate SOAP XML encodings.

3.3.1. Call

Example #1:

Following is an example of the SOAP encoding required to make this method call. This example uses the familiar HTTP verb POST. SOAP also supports the use of the HTTP verb M-POST for extensibility. See section 6.1 for more information on M-POST.

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml
Content-Length: nnnn
SOAPMethodName: Some-Namespace-URI#GetLastTradePrice

<SOAP:Envelope xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1">
    <SOAP:Body>
        <m:GetLastTradePrice
          xmlns:m="Some-Namespace-URI">
            <symbol>DIS</symbol>
        </m:GetLastTradePrice>
    </SOAP:Body>
</SOAP:Envelope>
```

3.3.2. Response

Example #2:

Following is the return message containing the HTTP headers and XML body:

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

<SOAP:Envelope xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1">
```

```
    <SOAP:Body>
        <m:GetLastTradePriceResponse xmlns:m="Some-Namespace-URI">
            <return>34.5</return>
        </m:GetLastTradePriceResponse>
    </SOAP:Body>
</SOAP:Envelope>
```

4. Relation to HTTP

In SOAP, the mechanism used for all communication is HTTP [1]. A central design goal of SOAP is that SOAP be usable strictly on top of today's deployed World Wide Web infrastructure. That means SOAP has to live with and work in the face of various levels of HTTP implementation, the active use of firewalls and proxies, and so on. Some aspects of SOAP, such as the permitted use of HTTP methods beyond those of classic HTTP, are designed to anticipate, and thus

make use of, some evolution and improvement in this base, but nothing in SOAP can require such fundamental changes in order for SOAP to function.

SOAP uses the Content-Type of "text/xml". This is used to specify the body of the HTTP request containing a XML encoded method call.

To disambiguate the headers it adds to HTTP, SOAP permits use of the HTTP Extension Framework specification [2].

Unless otherwise indicated in this document, existing practices with respect to the handling of HTTP requests and responses are to be adhered to. Specifically, this includes the following:

* Redirection
* Caching
* Connection management
* Support for access authentication and security

5. Relation to XML

XML is used to encode the call and response bodies. See [3] for more information on XML.

All protocol tags SHOULD be scoped to the SOAP namespace. The sender SHOULD include namespaces in SOAP elements and attributes. The receiver MUST correctly process SOAP payloads that have namespaces; the receiver MUST return an "Invalid Request" fault for requests that have incorrect namespaces. The receiver MAY also process SOAP payloads without namespaces as though they had the correct

namespaces.

The SOAP namespace has the proposed value "urn:schemas-xmlsoap-org:soap.v1". See [6] for more information on XML namespaces.

No XML document forming the HTTP request of a SOAP invocation may require the use of an XML DTD.

SOAP uses the ID attribute "id" to specify the unique identifier of an encoded element. SOAP uses the attribute "href" to specify a reference to that value, in a manner conforming to the XML Linking Language specification working draft [9].

It is worth noting that the rules governing XML payload format in SOAP are entirely independent of the fact that the payload is carried over an HTTP transport.

6. Method Invocation

A method invocation is performed by creating the HTTP request header and body and processing the returned response header and body. The request and response headers consist of standard and extended HTTP headers.

The following sections will cover the use of standard HTTP headers and the definition of extended HTTP headers.

6.1. HTTP Verb Rules

SOAP allows two verb options within the Call HTTP header: POST or M-POST.

The verb M-POST is an extension verb based on in the HTTP Extension Framework specification [2]. A SOAP invocation MUST first try the invocation by using POST.

If the POST invocation fails, it SHOULD retry using the HTTP method M-POST. The details of this mechanism are provided below. The purpose of supporting this extended invocation mechanism in SOAP is to provide a mechanism to unambiguously add headers to the HTTP protocol.

6.2. Using POST vs. M-POST

Since a design goal of the use of M-POST is to provide Internet firewalls and proxies greater administrative flexibility, careful

attention must be paid as to when a SOAP client uses the POST method vs. the M-POST method. The rules are as follows:

When carrying out an invocation, a SOAP client MUST first try the invocation using the POST invocation style.

If that POST invocation fails with an HTTP status of "405 Method Not Allowed" the client SHOULD retry the request using the M-POST invocation style. If that M-POST invocation fails with an HTTP status of "501 Not Implemented" or "510 Not Extended", the client SHOULD fail the request. If any other HTTP error is returned, it SHOULD be processed according to the HTTP specification.

Given this algorithm, firewalls can effectively force the use of M-POST for SOAP invocations by prohibiting POST invocations of Content-Type "text/xml" containing the HTTP header SOAPMethodName.

6.3. Method Invocation HTTP Headers

The payload and Content-Type of a method call are identical to a method response except in the following circumstances:

* The method call SHOULD contain additional HTTP header fields in the request:

a) If using the M-POST verb, a mandatory extension declaration MUST be present that refers to the namespace "urn:schemas-xmlsoap-org:soap.v1". For the purposes of this section, suppose that said declaration chooses to map the namespace to the header-prefix "01". If the POST verb is used, the namespace

header-prefix is not used. For example, the SOAPMethodName header introduced by SOAP would have an M-POST value of "01-SOAPMethodName" and a POST value of "SOAPMethodName".

b) The request SHOULD include a header "SOAPMethodName" whose value indicates the method to be invoked on the target. The value consists of a URI followed by a "#", followed by a method name (which MUST not include the "#" character). The URI used for the interface MUST match the implied or specified namespace qualification of the method name element in the SOAP:Body part of the payload. For example:

SOAPMethodName: http://electrocommerce.org/abc#MyMethod

* The server MUST return an "Invalid Request" SOAP:Fault if the required HTTP headers are missing or does not match the payload

exactly. Match means the URI and method name in the HTTP header exactly match the namespace URI AND LocalPart of the first element in the SOAP:Body.

6.4. Method Invocation Body

A SOAP method invocation consists of a method call and optionally a method response. The method call and method response are HTTP request and response respectively whose content is an XML document that consists of the root, (optional) header, and (mandatory) body elements. This XML document is referred to as SOAP payload in the rest of this specification.

The SOAP payload is defined as follows:

* The SOAP root element is the top element in the XML tree.
* The SOAP payload headers contain additional information that needs to travel with the call.
* The method request is represented as an XML element with additional elements for parameters. It is the first child of the SOAP:Body element.
* The response is the return value or error/exception that is passed back to the client.

The encoding rules are as follows:

1) Root element

a) The element tag is "SOAP:Envelope".

b) SOAP defines a global attribute "SOAP:encodingStyle" indicating any serialization rules used in lieu of those described by the SOAP spec. This attribute MAY appear on any element, and is scoped to that element and all child elements not themselves containing such an attribute. Omission of this attribute indicates that section 8 has been followed (unless overridden by a parent element). The URI "urn:schemas-xmlsoap-org:soap.v1"

is defined and all URIs beginning with this one indicate conformance with section 8 (though with potentially tighter rules).

c) The root element MAY contain namespace declarations.

d) The root element MAY contain additional attributes, provided these are namespace-qualified. The root element may contain additional sub elements provided these are namespace qualified

and follow the body.

2) SOAP payload headers

a) The element tag is "SOAP:Header". If present, this element MUST be the first element under the root.

b) It contains a list of header entries. Each MUST be namespace-qualified.

3) Call

a) The element tag is "SOAP:Body".

b) The Body element contains a first sub element whose name is the method name. This method request element in turn contains elements for each [in] and [in/out] parameter. The element names are the parameter names. See section 8 for details.

or

4) Response

a) The element tag is "SOAP:Body".

b) The Body element contains a first sub element that in turn contains child elements for each [in/out] and [out] parameter. The element names are the parameter names. See section 8 for details.

or

5) Fault

a) The element tag is "SOAP:Body".

b) The Body element contains a first sub element that is the SOAP:Fault element, indicating information about the fault.

The version of SOAP used is indicated by the SOAP namespace URI. A server MUST use the version passed in the envelope of the call for encoding the response, or it MUST fail the request. In the case where the server accepts a version or level less than its maximum, it MUST respond to the client by using the same version and level.

If a server receives a version it cannot handle, it must return a "Version Mismatch" SOAP:fault.

Processing requests in the face of missing parameters is application defined.

See section 7 for information on how to encode parameter values.

6.5. SOAP Payload Headers

In addition to the elements that specify direct, explicit information about the call or response, SOAP provides a way to pass extended, implicit information with the call through the use of the "header" element. It is encoded as a child of the SOAP:Envelope XML element. It contains a collection of distinctly named entries.

An example of the use of the header element is the passing of an implicit transaction ID along with a call. Since the transaction ID is not part of the signature and is typically held in an infrastructure component rather than application code, there is no direct way to pass the necessary information with the call. By adding an entry to the headers and giving it a fixed name, the transaction manager on the receiving side can extract the transaction ID and use it without affecting the coding of remote procedure calls.

Each header entry is encoded as an embedded element. The encoding rules for a header are as follows:

1. The element's name identifies the header. Header elements always are namespace-qualified.

2. Unless indicated to the contrary by the value of "SOAP:encodingStyle" attribute, header values are encoded according to the rules of section 8.

3. The element MAY contain an attribute "SOAP:mustUnderstand" specifying required understanding of the header by the destination.

An example is a header with an identifier of "TransactionID", a "mustUnderstand" value of true, and a value of 5. This would be encoded as follows:

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

6.5.1. The "SOAP:mustUnderstand" Attribute

Header entries MAY have a global attribute "SOAP:mustUnderstand". This may have one of two values, either "1" or "0". The absence of such a "SOAP:mustUnderstand" attribute is semantically equivalent to its presence with the value "0".

If a header element is tagged with a "SOAP:mustUnderstand" attribute whose value is "1", the party processing the Request URI MUST understand the semantics (as conveyed by its element tag, contextual setting, and so on) and process correctly to those semantics.

If the SOAP implementation doesn't understand the element, it MUST return an error as specified in section 7.1, "Results from a Method Call."

The SOAP:mustUnderstand attribute allows for robust semantic extensibility and change. Headers tagged with SOAP:mustUnderstand="1" MUST be presumed to somehow modify the semantics of their parent or peer elements. Tagging the headers in this manner assures that this change in semantics will not be silently (and, presumably, erroneously) ignored by those who may not fully understand it.

If the "SOAP:mustUnderstand" attribute is missing or has a value of "0", that element can safely be ignored.

For example: If the client passed along a transaction ID header, as in the above example, with a "SOAP:mustUnderstand" of "1", then the server SHOULD fail if it cannot process the transaction ID and comply with the transactional semantics.

6.6. Making a Method Call

To make a method call, the following information is needed:

* The URI of the target object
* A method name
* An optional method signature
* The parameters to the method
* Optional header data

The target URI of the HTTP request indicates the resource that the invocation is being made against; in this specification, that resource is called the "server address," to distinguish it from other uses of URIs. Other than it be a valid URI, SOAP places no restriction on the form of an address. See [8] for more information on URIs.

The body of a SOAP method call MUST be of Content-Type 'text/xml'.

The SOAP protocol places no absolute restriction on the syntax or case-sensitivity of interface names, method names, or parameter names. Of course, individual SOAP servers will respond to only the names they support; the selection of these is at their own sole

discretion. The one restriction is that the server MUST preserve the case of names.

6.6.1. Representation of Method Parameters

Method parameters are encoded as child elements of the call or response, encoded using the following rules:

1) The name of the parameter in the method signature is used as the name of the corresponding element.

2) Parameter values are expressed using the rules in section 8 of this document.

6.6.2. Sample Encoding of Call Requests

Example #3:

This sample is the same call as in section 3.3.1 but uses optional headers. It uses SOAP namespace to disambiguate SOAP keywords in the payload.

```
<SOAP:Envelope
  xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1"
  SOAP:encodingStyle="urn:schemas-xmlsoap-org:soap.v1">
     <SOAP:Header>
         <t:Transaction
           xmlns:t="some-URI"
           SOAP:mustUnderstand="1">
             5
         </t:Transaction>
     </SOAP:Header>
     <SOAP:Body>
         <m:GetLastTradePrice xmlns:m="Some-Namespace-URI">
             <symbol>DEF</symbol>
         </m:GetLastTradePrice>
     </SOAP:Body>
</SOAP:Envelope>
```

Example #4:

The following request sends a struct:

```
<SOAP:Envelope
  xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1" >
     <SOAP:Body>
         <m:GetLastTradePriceDetailed
           xmlns:m="Some-Namespace-URI">
             <Symbol>DEF</Symbol>
             <Company>DEF Corp</Company>
             <Price> 34.1 </Price>
         </m:GetLastTradePriceDetailed>
     </SOAP:Body>
```



```
</SOAP:Envelope>
```

7. Results of Method Calls

At the receiving site, a call request can have one of the following four outcomes:

a) The HTTP infrastructure on the receiving site was able to receive and process the request.

b) The HTTP infrastructure on the receiving site could not receive and process the request.

c) The SOAP infrastructure on the receiving site was able to decode the input parameters, dispatch to an appropriate server indicated by the server address, and invoke an application-level function corresponding semantically to the method indicated in the method call.

d) The SOAP infrastructure on the receiving site could not decode the input parameters, dispatch to an appropriate server indicated by the server address, and invoke an application-level function corresponding semantically to the interface or method indicated in the method call.

In the first case, the HTTP infrastructure passes the headers and body to the SOAP infrastructure.

In the second case, the result is an HTTP response containing an HTTP error in the status field and no XML body.

In the third case, the result of the method call consists of a response or fault.

In the fourth case, the result of the method is a fault indicating a fault that prevented the dispatching infrastructure on the receiving side from successful completion.

In the third and fourth cases, additional payload headers MAY for extensibility again be present in the results of the call.

7.1. Results from a Method Call

The results of the call are to be provided in the form of a call response. The HTTP response MUST be of Content-Type "text/xml".

Because a result indicates success and a fault indicates failure, it is an error for the method response to contain both a result and a fault.

7.2. SOAP:Fault and HTTP Status Codes

If the HTTP infrastructure successfully processes the method request, passes it to the SOAP infrastructure, and an error occurs, a fault indication is returned to the caller instead of a normal response. In this section, a SOAP:Fault element is defined. This element MUST be used to return fault indications.

The standard SOAP:Fault element consists of four sub elements:

* "faultcode", which MUST contain a qualified name value, as defined in Namespaces in XML [6], section 3 "Qname". If unqualified, the value is taken from the space of SOAP status codes, described below. If qualified, the prefix MUST match a declared namespace prefix. The faultcode is intended for use by software.

* "faultstring", which MUST contain a string value. The faultstring is intended for use by human users and must not be acted upon algorithmically by software. faultstring is similar to the 'Reason-Phrase' that may be present in HTTP responses. (See [1], section 6.1.)

* "runcode", which MUST contain an enumerated value. The runcode is intended to indicate whether or not the request was dispatched to the application. There are three runcodes currently defined: "Maybe", "No", "Yes".

* "detail", which if present MUST contain a value per section 8 with application-specific semantics.

Other sub elements members beyond the three described above MAY be present, provided they are namespace-qualified.

If the fault specifies a server fault, as opposed to an HTTP fault, the HTTP status code MUST be "200" and the HTTP status message MUST be "OK". If it specifies an HTTP fault, the HTTP status code as defined in the HTTP specification [1] SHOULD be used.

If a method call fails to be processed because of a non-understood extension header element contained therein, the method invocation MUST return a fault. The fault MUST contain a 'faultcode' of "Must Understand".

If a method response fails to be processed for similar reasons, an appropriate exceptional condition should be indicated to the application layer in an implementation-defined manner.

7.3. SOAP Status Codes

SOAP defines its own space of status codes. This space is used only by the SOAP infrastructure and is not expected to be used on HTTP failure. The reason this space is defined is to aid the conversion of existing protocols onto SOAP.

This status code space MUST be used for faultcodes contained in faults and in the method definitions defined in this specification that return status code values. Further, use of this space is recommended (but not required) in the specification of methods defined outside of the present specification.

The SOAP status code space is identified by the URI, "urn:schemas-xmlsoap-com:soap.v1/faultcode" and contains numeric values drawn from the following ranges:

This specification at present defines the following status codes beyond those specified in [1]:

Name	Value	Meaning
Version Mismatch	100	The call was using an unsupported SOAP version.
Must Understand	200	An XML element was received that contained

		an element tagged with mustUnderstand="1" that was not understood by the receiver.
Invalid Request	300	The receiving application did not process the request because it was ill formed or not supported by the application.
Application Faulted	400	The receiving application faulted processing the request. The 'detail' element contains the application specific fault.

7.4. Sample Encoding of Response

Example #5:

The response from the example in section 3.3.2 would be:

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

<SOAP:Envelope
  xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1" >
     <SOAP:Body>
         <m:GetLastTradePriceResponse
           xmlns:m="Some-Namespace-URI">
```



```
            <return>34.5</return>
         </m:GetLastTradePriceResponse>
     </SOAP:Body>
</SOAP:Envelope>
```

Example #6:

The following response is similar to the one above, but uses optional headers.

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

```
<SOAP:Envelope
  xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1" >
     <SOAP:Header>
         <t:Transaction
           xmlns:t="some-URI"
           xsd:type="int" mustUnderstand="1">
             5
         </t:Transaction>
     </SOAP:Header>
     <SOAP:Body>
         <m:GetLastTradePriceResponse
           xmlns:m="Some-Namespace-URI">
             <return>34.5</return>
         </m:GetLastTradePriceResponse>
     </SOAP:Body>
</SOAP:Envelope>
```

Example #7:

The following response returns a struct:

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

<SOAP:Envelope
  xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1" >
    <SOAP:Body>
        <m:GetLastTradePriceResponse
          xmlns:m="Some-Namespace-URI">
            <return>
                <LastTradePrice>
                    34.5
                </LastTradePrice>
                <DayVolume>
                    10000
                </DayVolume>
            </return>
        </m:GetLastTradePriceResponse>
```



```
    </SOAP:Body>
</SOAP:Envelope>
```

Example #8:

If there was an error in the HTTP infrastructure, the response could

be as follows:

```
HTTP/1.1 401 Unauthorized
```

Example #9:

If there was an error in the SOAP infrastructure processing the request on the server, the response could be as follows:

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

<SOAP:Envelope xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1">
    <SOAP:Body>
        <SOAP:Fault>
            <faultcode>200</faultcode>
            <faultstring>
               SOAP Must Understand Error
            </faultstring>
            <runcode>1</runcode>
        </SOAP:Fault>
    <SOAP:Body>
</SOAP:Envelope>
```

Example #10:

If the application passed back its own fault element, the response would be as follows:

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

<SOAP:Envelope
  xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1" >
    <SOAP:Body>
        <SOAP:Fault>
            <faultcode>400</faultcode>
            <faultstring>
                SOAP Must Understand Error
            </faultstring>
            <runcode>1</runcode>
            <detail xmlns:e="Some-Namespace-URI"
                    xmlns:xsd="W3C-Schemas-URI"
                    xsd:type="e:MyFault"
                <message>
```

```
                  My application didn't work
                </message>
                <errorcode>1001</errorcode>
            </detail>
        </SOAP:Fault>
    </SOAP:Body>
</SOAP:Envelope>
```

8. Types

SOAP uses a simple, traditional type system. A type either is a simple (scalar) type or is a compound type constructed as a composite of several parts, each with a type.

Because all types are contained or referenced within a call or response element, the encoding samples in this section assume all namespace declarations are at a higher element level.

8.1. Rules for Encoding Types in XML

XML allows very flexible encoding of data to represent a method call. SOAP defines a narrower set of rules for encoding. This section defines the encoding rules at a high level, and the next section describes the encoding rules for specific types when they require more detail.

To describe encoding, the following terminology is used:

1. A "type" includes integer, string, point, or street address. A type in SOAP corresponds to a scalar or structured type in a programming language or database. All values are of specific types.

2. A "compound type" is one that has distinct, named parts and whose encoding should reflect those named parts. A "simple type" is one without named parts. A structured type in a programming language is a compound type, and so is an array.

3. The name of a parameter or of a named part of a compound type is called an "accessor."

4. If only one accessor can reference it, a value is considered "single-reference" for a given schema. If referenced by more than one, actually or potentially in a given schema, it is "multi-reference." Therefore, it is possible for a certain type to be considered "single-reference" in one schema and "multi-reference" in another schema.

5. Syntactically, an element MAY be "independent" or "embedded." An independent element is contained immediately by its scoping element. An embedded element is contained within a non-scoping element. Examples of scoping element in this specification are

"SOAP:Header" and "SOAP:Body".

The rules are as follows:

1. Elements MAY be used to reflect either accessors or instances of types. Embedded elements always reflect accessors. Independent elements always reflect instances of types. When reflecting an accessor, the name of the element gives the name of the accessor. When reflecting an instance of a type, the name of the element typically gives the name of the type.

2. A call or response/fault is always encoded as an independent element.

3. Accessors are always encoded as embedded elements.

4. A value (simple or compound) is encoded as element content, either of an element reflecting an accessor to the value or of an element reflecting an instance of that type.

5. A simple value is encoded as character data, that is, without any sub elements.

6. Strings and byte arrays are multi-reference simple types, but special rules allow them to be represented efficiently for common cases. An accessor to a string or byte-array value MAY have an attribute named "id" and of type "ID" per the XML Specification [3]. If so, all other accessors to the same value are encoded as empty elements having an attribute named "href" and of type "URI" per the XML Linking Language Specifications [9], with the href containing a URI fragment identifier referencing the single element containing the value.

7. It is permissible to encode several references to a simple value as though these were references to several single-reference values, but only when from context it is known that the meaning of the XML instance is unaltered.

8. A compound value is encoded as a sequence of elements, each named according to the accessor it reflects. (See also section 8.4.1.)

9. A multi-reference simple or compound value is encoded as an independent element containing an attribute named "id" and of type "ID" per the XML Specification [3]. Each accessor to this value is an empty element having an attribute named "href" and of type "URI" per the XML Linking Language Specification [9], with the href

containing a URI fragment identifier referencing the corresponding independent element.

10. Arrays are compound types. Arrays can be of one or more dimensions (rank) whose elements are normally laid contiguously in memory. Arrays can be single-reference or multi-reference values. Single-reference embedded arrays are encoded using accessor elements. A multi-reference array is always encoded as an independent element whose tag name is the string "ArrayOf" prepended

to the element type of the array. The independent element or the accessor MUST contain a "xsd:type" attribute that specifies the type and dimensions of the array and is encoded as the type of the array element, followed by "[", followed by comma-separated lengths of each dimension, followed by "]". The "xsd:type" attribute is described in the "XML Schema Part 2: Datatypes" Specification (see 12]. Note that the array element itself can be an array. An array type is encoded as its element type, followed by "[", followed by rank encoded as a sequence of commas(one for each dimension), followed by "]". It MAY also contain an "offset" attribute to indicate the starting position of a partially represented array. Each element of an array is encoded using the accessor named after the type of array element. The elements are represented as a list with the dimension on the right side varying rapidly. The accessor MAY contain the "position" attribute that conveys the position of the item in the enclosing array. Both "offset" and "position" attributes are encoded as "[", followed by a comma-separated position in each dimension, followed by "]", with offsets and positions based at 0.

11. Any accessor element that contains its value directly MAY optionally have an attribute named "xsd:type" whose value indicates the type of the element's contained value as described in the "XML Schema Part 2: Datatypes" Specification [12]. However, its presence is mandatory on elements whose tag name combined with implicit or explicit namespace does not unambiguously identify the type of the element.

12. A NULL value is indicated by an attribute named "xsd:null" with value of '1'.

13. In many cases, it is necessary to package multiple related elements as a single element, for instance in cases where the elements are linked together via hrefs. SOAP defines another attribute "SOAP:Package" whose value can be "0" or "1". If "1" the element acts as a scoping element for contained sub elements.

8.2. Simple Types

For simple types, SOAP adopts the types found in the section "Built-in datatypes" of the "XML Schema Part 2: Datatypes" Specification [12], along with the corresponding recommended representation thereof. Examples include:

```
integer:             58502
real:                314159265358979E+1
negative-integer:    -32768
```

Strings and arrays of bytes are encoded as multi-reference simple types.

8.2.1. String

A string is a multi-reference simple type. According to the rules of multi-reference simple types, the containing element of the string value MAY have an ID attribute; additional accessor elements MAY then have matching href attributes.

For example, two accessors to the same string could appear, as follows:

```
<greeting id="String-0">Hello</greeting>
<salutation href="#String-0"/>
```

However, if the fact that both accessors reference the same instance of the string is immaterial, they may be encoded as though single-reference, as follows:

```
<greeting>Hello</greeting>
<salutation>Hello</salutation>
```

8.2.2. Enums

An enum is a single reference type whose value is encoded as one of the possible enumeration strings. In the following example EyeColor is an enum with the possible values of "Green", "Blue", and "Brown":

```
<Person>
     <Name>Henry Ford</Name>
     <Age>32</Age>
     <EyeColor>Brown</EyeColor>
</Person>
```

8.2.3. Array of Bytes

An array of bytes is encoded as a multi-reference simple type. The recommended representation of an opaque array of bytes is the 'bin.base64' encoding defined in XML DCD [5], which simply references the MIME standard. However, the line length restrictions that normally apply to Base64 data in MIME do not apply in SOAP.

bin.base64: aG93IG5vdyBicm93biBjb3cNCg==

8.3. Polymorphic Accessor

Many languages allow accessors that can polymorphically access values of several types, each type being available at run-time. When the value is single-reference, the type of this kind of accessor is often called "Variant". A Polymorphic accessor MUST contain a "xsd:type" attribute that describes the type of the actual value.

For example, a Polymorphic parameter named "cost" with a type of float would be encoded as follows:

```
<cost xsd:type="float">29.95</cost>
```


as contrasted with a cost parameter whose type is invariant, as follows:

```
<cost>29.95</cost>
```

8.4. Compound Types

Beyond the simple types, SOAP defines support for the following constructed types:

* Records/structs
* arrays

Where appropriate and possible, the representation in SOAP of a value of a given type mirrors that used by practitioners of XML-Data and the common practice of the XML community at large.

8.4.1. Compound Values and References to Values

A compound value contains an ordered sequence of structural members. When the members have distinct names, as in an instance of a C or C++ "struct", this is called a "struct," and when the members do not

have distinct names but instead are known by their ordinal position, this is called an "array.

The members of a compound value are encoded as accessor elements. For a struct, the accessor element name is the member name. For an array, the accessor element name is the element type name and the sequence of the accessor elements follows the ordinal sequence of the members.

The following is an example of a struct of type Book:

```
<Book>
     <author>Henry Ford</author>
     <preface>Prefatory text</preface>
     <intro>This is a book.</intro>
</Book>
```

Below is an example of a type with both simple and compound members. It shows two levels of referencing.

Note that the "href" attribute of the Author accessor element is a reference to the value whose "id" attribute matches; a similar construction appears for the Address.

```
<Book>
     <title>My Life and Work</title>
     <author href="#Person-1"/>
</Book>
<Person id="Person-1">
     <name>Henry Ford</name>
     <address href="#Address-2"/>
```



```
</Person>
<Address id="Address-2">
     <email>henryford@hotmail.com</email>
     <web>www.henryford.com</web>
</Address>
```

The form above is appropriate when the Person value and the Address value are multi-reference. If these were instead both single-reference, they SHOULD be embedded, as follows:

```
<Book>
     <title>My Life and Work</title>
     <author>
          <name>Henry Ford</name>
          <address>
```

```
                <email>henryford@hotmail.com</email>
                <web>www.henryford.com</web>
            </address>
        </author>
    </Book>
```

If instead there existed a restriction that no two persons can have the same address in a given schema and that an address can be either a Street-address or an Electronic-address, a Book with two authors would be encoded in such a schema as follows:

```
    <Book>
        <title>My Life and Work</title>
        <firstauthor href="#Person-1"/>
        <secondauthor href="#Person-2"/>
    </Book>
    <Person id="Person-1">
        <name>Henry Ford</name>
        <address xsd:type="m:Electronic-address">
            <email>henryford@hotmail.com</email>
            <web>www.henryford.com</web>
        </address>
    </Person>
    <Person id="Person-2">
        <name>Thomas Cook</name>
        <address xsd:type="n:Street-address">
            <Street>Martin Luther King Rd</Street>
            <City>Raleigh</City>
            <State>North Carolina</State>
        </address>
    </Person>
```

8.4.1.1. Generic Records

There are cases where a struct is represented with its members named and values typed at run time. Even in these cases, the existing rules apply. Each member is encoded as an element with matching name, and each value is either contained or referenced. Contained

values MUST have a "xsd:type" attribute giving the type of the value.

8.4.2. Arrays

The representation of the value of an array is an ordered sequence of elements constituting items of the array. The default tag name for each element is the element type.

As with compound types generally, if the type of an item in the array is a single-reference type, each item contains its value. Otherwise, the item references its value via an href attribute.

The following example is an array containing integer array members. The length attribute is optional.

```
<ArrayOfint xsd:type="u:int[2]">
     <int>3</int>
     <int>4</int>
</ArrayOfint>
```

The following example is an array of Variants containing an integer and a string.

```
<ArrayOfvariant xsd:type="u:variant[2]">
     <variant xsd:type="int">23</variant>
     <variant xsd:type="string">some string</variant>
</ArrayOfvariant>
```

The following is an example of a two-dimensional array of strings.

```
<ArrayOfstring xsd:type="u:string[2,3]">
     <string>r1c1</string>
     <string>r1c2</string>
     <string>r1c3</string>
     <string>r2c1</string>
     <string>r2c2</string>
     <string>r2c3</string>
</ArrayOfstring>
```

The following is an example of an array of two arrays, each of which is an array of strings.

```
<ArrayOfArrayOfstring xsd:type="u:string[][2]">
     <ArrayOfstring href="#array-1"/>
     <ArrayOfstring href="#array-2"/>
</ArrayOfArrayOfstring>
<ArrayOfstring id="array-1" xsd:type="u:string[3]">
     <string>r1c1</string>
     <string>r1c2</string>
     <string>r1c3</string>
</ArrayOfstring>
<ArrayOfstring id="array-2" xsd:type="u:string[2]">
```



```
     <string>r2c1</string>
```

```
     <string>r2c2</string>
</ArrayOfstring>
```

Finally, the following is an example of an array of phone numbers embedded in a struct of type Person and accessed through the accessor "phone-numbers":

```
<Person>
     <name>John Hancock</name>
     <phone-numbers xsd:type="u:string[2]">
             <string>111-2222</string>
             <string>999-0000</string>
     </phone-numbers>
</Person>
```

A multi-reference array is always encoded as an independent element whose tag name is the string "ArrayOf" prepended to the element type of the array. For example an array of order structs encoded as an independent element:

```
<ArrayOfOrder xsd:type="u:Order[2]">
     <Order>
          <Product>Apple</Product>
          <Price>1.56</Price>
     </Order>
     <Order>
          <Product>Peach</Product>
          <Price>1.48</Price>
     </Order>
</ArrayOfOrder>
```

A single-reference array is encoded as an embedded element whose tag name is the accessor name.

```
<PurchaseOrder>
     <CustomerName>Henry Ford</CustomerName>
     <ShipTo>
          <Street>5th Ave</Street>
          <City>New York</City>
          <State>NY</State>
          <Zip>10010</Zip>
     </ShipTo>
     <PurchaseLineItems xsd:type="u:Order[2]">
             <Order>
                     <Product>Apple</Product>
                     <Price>1.56</Price>
             </Order>
             <Order>
                     <Product>Peach</Product>
                     <Price>1.48</Price>
             </Order>
```

```
        </PurchaseLineItems>
```



```
   </PurchaseOrder>
```

8.4.2.1. Partially transmitted arrays

SOAP provides support for partially transmitted arrays, known as "varying" arrays, in some contexts [7]. A partially transmitted array indicates in an "offset" attribute the zero-origin index of the first element transmitted; if omitted, the offset is taken as zero.

The following is an example of an array of size five that transmits only the third and fourth element:

```
<ArrayOfstring xsd:type="u:string[5]" offset="[2]">
     <string>The third element</string>
     <string>The fourth element</string>
</Arrayofstring>
```

8.4.2.2. Sparse Arrays

SOAP provides support for sparse arrays in some contexts. Each element contains a "position" attribute that indicates its position within the array. The following is an example of array of arrays of strings:

```
<ArrayOfArrayOfstring xsd:type="u:string[,][2]">
     <Arrayofstring href="#array-1" position="[2]"/>
</ArrayOfArrayOfstring>

<ArrayOfstring id="array-1" xsd:type="u:string[10,10]">
     <string position="[2,2]">The third element"</item>
     <string position="[7,2]">The eighth element</item>
</ArrayOfstring>
```

Assuming that the only reference to array-1 occurs in the enclosing array, this example could also have been encoded as follows:

```
<ArrayOfArrayOfstring xsd:type="string[,][2]">
     <ArrayOfstring position="[2]">
          <ArrayOfstring xsd:type="string[10,10]">
               <string position="[2,2]">The third element"</string>
               <string position="[7,2]">The eighth element</string>
          </ArrayOfstring>
     </ArrayOfstring>
</ArrayOfArrayOfstring>
```

8.5. Default Values

An omitted accessor element implies either a default value or that no value is known. The specifics depend on the accessor, method, and its context. Typically, an omitted accessor implies a Null value for Variant and for polymorphic accessors (with the exact meaning of Null accessor-dependent). Typically, an omitted Boolean accessor

implies either a False value or that no value is known, and an omitted numeric accessor implies either that the value is zero or that no value is known.

9. Formal Syntax

This specification uses the augmented Backus-Naur Form (BNF) as described in RFC-2234 [10].

10. Security Considerations

Not described in this document are methods for integrity and privacy protection. Such issues will be addressed more fully in a future version(s) of this document.

11. References

[1] RFC2068: Hypertext Transfer Protocol, http://info.internet.isi.edu/in-notes/rfc/files/rfc2068.txt. Also: http://www.w3.org/Protocols/History.html.
[2] HTTP Extension Framework, http://www.w3.org/Protocols/HTTP/ietf-http-ext.
[3] The XML Specification, http://www.w3.org/TR/WD-xml-lang.
[4] XML-Data Specification, http://www.w3.org/TR/1998/NOTE-XML-data.
[5] Document Content Description for XML, http://www.w3.ort/TR/NOTE-dcd.
[6] Namespaces in XML, http://www.w3.org/TR/REC-xml-names.
[7] Transfer Syntax NDR, in "DCE 1.1: Remote Procedure Call," http://www.rdg.opengroup.org/onlinepubs/9629399/toc.htm.
[8] RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax and Semantics, http://www.ietf.org/rfc/rfc2396.txt.
[9] XML Linking Language, http://www.w3.org/1999/07/WD-xlink-19990726.
[10] RFC-2234: Augmented BNF for Syntax Specifications: ABNF
[11] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997
[12] XML Schema Part 2: Datatypes, http://www.w3.org/TR/1999/WD-

xmlschema-2-19991105/
[13] RFC2026: The Internet Standards Process -- Revision 3, http://info.internet.isi.edu/in-notes/rfc/files/rfc2026.txt.

12. Author's Addresses

Don Box
DevelopMentor
21535 Hawthorne Blvd., Fourth Floor
Torrance, CA 90503
Email: dbox@develop.com

Gopal Kavivaya
Microsoft
One Microsoft Way

Redmond, WA 98052
Email: gopalk@microsoft.com

Andrew Layman
Microsoft
One Microsoft Way
Redmond, WA 98052
Email: andrewl@microsoft.com

Satish Thatte
Microsoft
One Microsoft Way
Redmond, WA 98052
Email: satisht@microsoft.com

Dave Winer
UserLand Software, Inc.
P.O. Box 1218
Burlingame, CA 94011-1218
Email: dave@userland.com

Full Copyright Statement