SOAP Version 1.2 Part 0: Primer

W3C Recommendation 24 June 2003

This version:
http://www.w3.org/TR/2003/REC-soap12-part0-20030624/
Latest version:
http://www.w3.org/TR/soap12-part0/
Previous version:
http://www.w3.org/TR/2003/PR-soap12-part0-20030507/
Editor:
Nilo Mitra (Ericsson)

Please refer to the **errata** for this document, which may include some normative corrections.

The English version of this specification is the only normative version. Non-normative translations may also be available.

Copyright © 2003 W3C® (MIT, ERCIM, Keio), All Rights Reserved. W3C liability, trademark, document use, and software licensing rules apply.

Abstract

SOAP Version 1.2 Part 0: Primer is a non-normative document intended to provide an easily understandable tutorial on the features of the SOAP Version 1.2 specifications. In particular, it describes the features through various usage scenarios, and is intended to complement the normative text contained in Part 1 and Part 2 of the SOAP 1.2 specifications.

Status of this Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. The latest status of this document series is maintained at the W3C.

This document is a Recommendation of the W3C. This document has been produced by the XML Protocol Working Group, which is part of the Web Services Activity. It has been reviewed by W3C Members and other interested parties, and has been endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited as a normative reference from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

Comments on this document are welcome. Please send them to the public mailing-list xmlp-comments@w3.org (archive). It is inappropriate to send discussion email to this address.

Information about implementations relevant to this specification can be found in the Implementation Report at http://www.w3.org/2000/xp/Group/2/03/soap1.2implementation.html.

Patent disclosures relevant to this specification may be found on the Working Group's patent disclosure page, in conformance with W3C policy.

A list of current W3C Recommendations and other technical reports can be found at http://www.w3.org/TR.

Table of Contents

1. Introduction
    1.1 Overview
    1.2 Notational Conventions
2. Basic Usage Scenarios
    2.1 SOAP Messages
    2.2 SOAP Message Exchange
        2.2.1 Conversational Message Exchanges
        2.2.2 Remote Procedure Calls
    2.3 Fault Scenarios
3. SOAP Processing Model
    3.1 The "role" Attribute
    3.2 The "mustUnderstand" Attribute
    3.3 The "relay" Attribute
4. Using Various Protocol Bindings
    4.1 The SOAP HTTP Binding
        4.1.1 SOAP HTTP GET Usage
        4.1.2 SOAP HTTP POST Usage
        4.1.3 Web Architecture Compatible SOAP Usage
    4.2 SOAP Over Email
5. Advanced Usage Scenarios
    5.1 Using SOAP Intermediaries
    5.2 Using Other Encoding Schemes
6. Changes Between SOAP 1.1 and SOAP 1.2
7. References
A. Acknowledgements

1. Introduction

SOAP Version 1.2 Part 0: Primer is a non-normative document intended to provide an easily understandable tutorial on the features of the SOAP Version 1.2 specifications. Its purpose is to help a technically competent person understand how SOAP may be used, by describing representative SOAP message structures and message exchange patterns.

In particular, this primer describes the features of SOAP through various usage scenarios, and is intended to complement the normative text contained in SOAP Version 1.2 Part 1: Messaging Framework (hereafter [SOAP Part1]) and SOAP Version 1.2 Part 2: Adjuncts (hereafter [SOAP Part2]) of the SOAP Version 1.2 specifications.

It is expected that the reader has some familiarity with the basic syntax of XML, including the use of XML namespaces and infosets, and Web concepts such as URIs and HTTP. It is intended primarily for users of SOAP, such as application designers, rather than implementors of the SOAP specifications, although the latter may derive some benefit. This primer aims at highlighting the essential features of SOAP Version 1.2, not at completeness in describing every nuance or edge case. Therefore, there is no substitute for the main specifications to obtain a fuller understanding of SOAP. To that end, this primer provides extensive links to the main specifications wherever new concepts are introduced or used.

[SOAP Part1] defines the SOAP envelope, which is a construct that defines an overall framework for representing the contents of a SOAP message, identifying who should deal with all or part of it, and whether handling such parts are optional or mandatory. It also defines a protocol binding framework, which describes how the specification for a binding of SOAP onto another underlying protocol may be written.

[SOAP Part2] defines a data model for SOAP, a particular encoding scheme for data types which may be used for conveying remote procedure calls (RPC), as well as one concrete realization of the underlying protocol binding framework defined in [SOAP Part1]. This binding allows the exchange of SOAP messages either as payload of a HTTP POST request and response, or as a SOAP message in the response to a HTTP GET.

This document (the primer) is not normative, which means that it does not provide the definitive specification of SOAP Version 1.2. The examples provided here are intended to complement the formal specifications, and in any question of interpretation the formal specifications naturally take precedence. The examples shown here provide a subset of the uses expected for SOAP. In actual usage scenarios, SOAP will most likely be a part of an overall solution, and there will no doubt be other application-specific requirements which are not captured in these examples.

1.1 Overview

SOAP Version 1.2 provides the definition of the XML-based information which can be used for exchanging structured and typed information between peers in a decentralized, distributed environment. [SOAP Part1] explains that a SOAP message is formally specified as an XML Infoset [XML Infoset], which provides an abstract description of its contents. Infosets can have different on-the-wire representations, one common example of which is as an XML 1.0 [XML 1.0] document.

SOAP is fundamentally a stateless, one-way message exchange paradigm, but applications can create more complex interaction patterns (e.g., request/response, request/multiple responses, etc.) by combining such one-way exchanges with features provided by an underlying protocol and/or application-specific information. SOAP is silent on the semantics of any application-specific data it conveys, as it is on issues such as the routing of SOAP messages, reliable data transfer, firewall traversal, etc. However, SOAP provides the framework by which application-specific information may be conveyed in an extensible manner. Also, SOAP provides a full description of the required actions taken by a SOAP node on receiving a SOAP message.

Section 2 of this document provides an introduction to the basic features of SOAP starting with the simplest usage scenarios, namely a one-way SOAP message, followed by various request-response type exchanges, including RPCs. Fault situations are also described.

Section 3 provides an overview of the SOAP processing model, which describes the rules for initial construction of a message, rules by which messages are processed when received at an intermediary or ultimate destination, and rules by which portions of the message can be inserted, deleted or modified by the actions of an intermediary.

Section 4 of this document describes the ways in which SOAP messages may be transported to realize various usage scenarios. It describes the SOAP HTTP binding specified in [SOAP Part2], as well as an example of how SOAP messages may be conveyed in email messages. As a part of the HTTP binding, it introduces two message exchange patterns which are available to an application, one of which uses the HTTP POST method, while the other uses HTTP GET. Examples are also provided on how RPCs, in particular those that represent "safe" information retrieval, may be represented in SOAP message exchanges in a manner that is compatible with the architectural principles of the World Wide Web .

Section 5 of this document provides a treatment of various aspects of SOAP that can be used in more complex usage scenarios. These include the extensibility mechanism offered through the use of header elements, which may be targeted at specific intermediate SOAP nodes to provide value-added services to communicating applications, and using various encoding schemes to serialize application-specific data in SOAP messages.

Section 6 of this document describes the changes from SOAP Version 1.1 [SOAP 1.1].

Section 7 of this document provides references.

For ease of reference, terms and concepts used in this primer are hyper-linked to their definition in the main specifications.

1.2 Notational Conventions

Throughout this primer, sample SOAP envelopes and messages are shown as [XML 1.0] documents. [SOAP Part1] explains that a SOAP message is formally specified as an [XML InfoSet], which is an abstract description of its contents. The distinction between the SOAP XML Infosets and the documents is unlikely to be of interest to those using this primer as an introduction to SOAP; those who do care (typically those who port SOAP to new protocol bindings where the messages may have alternative representations) should understand these examples as referring to the corresponding XML infosets. Further elaboration of this point

is provided in Section 4 of this document.

The namespace prefixes "env", "enc", and "rpc" used in the prose sections of this document are associated with the SOAP namespace names "http://www.w3.org/2003/05/soap-envelope" , "http://www.w3.org/2003/05/soap-encoding", and "http://www.w3.org/2003/05/soap-rpc" respectively.

The namespace prefixes "xs" and "xsi" used in the prose sections of this document are associated with the namespace names "http://www.w3.org/2001/XMLSchema" and "http://www.w3.org/2001/XMLSchema-instance" respectively, both of which are defined in the XML Schema specifications [XML Schema Part1], [XML Schema Part2].

Note that the choice of any other namespace prefix is arbitrary and not semantically significant.

Namespace URIs of the general form "http://example.org/..." and "http://example.com/..." represent an application-dependent or context-dependent URI [RFC 2396].

2. Basic Usage Scenarios

A SOAP message is fundamentally a one-way transmission between SOAP nodes, from a SOAP sender to a SOAP receiver, but SOAP messages are expected to be combined by applications to implement more complex interaction patterns ranging from request/response to multiple, back-and-forth "conversational" exchanges.

The primer starts by exposing the structure of a SOAP message and its exchange in some simple usage scenarios based on a travel reservation application. Various aspects of this application scenario will be used throughout the primer. In this scenario, the travel reservation application for an employee of a company negotiates a travel reservation with a travel booking service for a planned trip. The information exchanged between the travel reservation application and the travel service application is in the form of SOAP messages.

The ultimate recipient of a SOAP message sent from the travel reservation application is the travel service application, but it is possible that the SOAP message may be "routed" through one or more SOAP intermediaries which act in some way on the message. Some simple examples of such SOAP intermediaries might be ones that log, audit or, possibly, amend each travel request. Examples, and a more detailed discussion of the behavior and role of SOAP intermediaries, is postponed to section 5.1.

Section 2.1 describes a travel reservation request expressed as a SOAP message, which offers the opportunity to describe the various "parts" of a SOAP message.

Section 2.2.1 continues the same scenario to show a response from the travel service in the form of another SOAP message, which forms a part of a conversational message exchange as the various choices meeting the constraints of the travel request are negotiated.

Section 2.2.2 assumes that the various parameters of the travel reservation have been accepted by the traveller, and an exchange - modelled as a remote procedure call (RPC) - between the travel reservation and the travel service applications confirms the payment for the reservation.

Section 2.3 shows examples of fault handling.

2.1 SOAP Messages

Example 1 shows data for a travel reservation expressed in a SOAP message.

Example 1

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
          env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
   <m:dateAndTime>2001-11-29T13:20:00.000-05:00</m:dateAndTime>
```

```
   </m:reservation>
   <n:passenger xmlns:n="http://mycompany.example.com/employees"
          env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
    <n:name>Åke Jógvan Øyvind</n:name>
   </n:passenger>
  </env:Header>
  <env:Body>
   <p:itinerary
     xmlns:p="http://travelcompany.example.org/reservation/travel">
    <p:departure>
      <p:departing>New York</p:departing>
      <p:arriving>Los Angeles</p:arriving>
      <p:departureDate>2001-12-14</p:departureDate>
      <p:departureTime>late afternoon</p:departureTime>
      <p:seatPreference>aisle</p:seatPreference>
    </p:departure>
    <p:return>
      <p:departing>Los Angeles</p:departing>
      <p:arriving>New York</p:arriving>
      <p:departureDate>2001-12-20</p:departureDate>
      <p:departureTime>mid-morning</p:departureTime>
      <p:seatPreference/>
    </p:return>
   </p:itinerary>
   <q:lodging
    xmlns:q="http://travelcompany.example.org/reservation/hotels">
    <q:preference>none</q:preference>
   </q:lodging>
  </env:Body>
 </env:Envelope>
```

Sample SOAP message for a travel reservation containing header blocks and a body

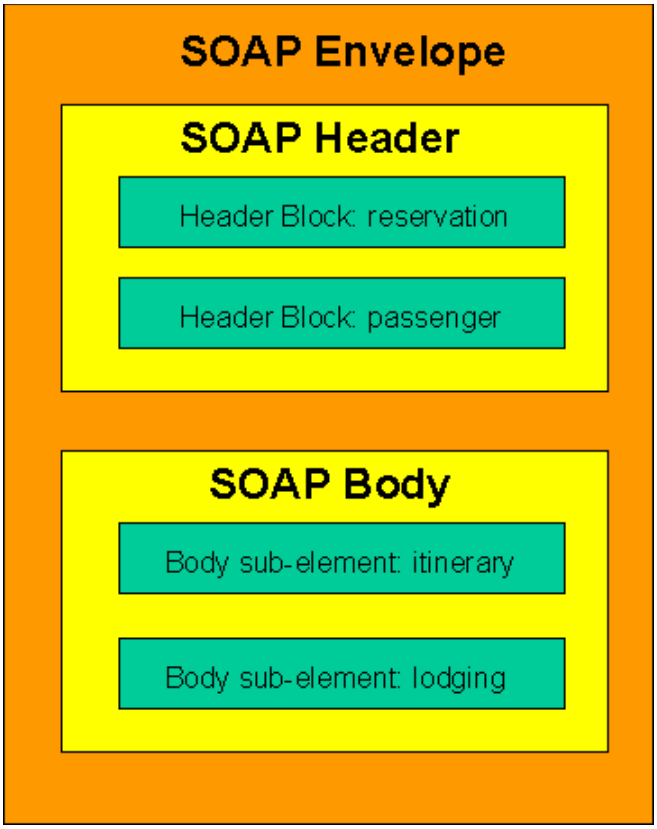
The SOAP message in Example 1 contains two SOAP-specific sub-elements within the overall `env:Envelope`, namely an `env:Header` and an `env:Body`. The contents of these elements are application defined and not a part of the SOAP specifications, although the latter do have something to say about how such elements must be handled.

A SOAP header element is optional, but it has been included in the example to explain certain features of SOAP. A SOAP header is an extension mechanism that provides a way to pass information in SOAP messages that is not application payload. Such "control" information includes, for example, passing directives or contextual information related to the processing of the message. This allows a SOAP message to be extended in an application-specific manner. The immediate child elements of the `env:Header` element are called header blocks, and represent a logical grouping of data which, as shown later, can individually be targeted at SOAP nodes that might be encountered in the path of a message from a sender to an ultimate receiver.

SOAP headers have been designed in anticipation of various uses for SOAP, many of which will involve the participation of other SOAP processing nodes - called SOAP intermediaries - along a message's path from an initial SOAP sender to an ultimate SOAP receiver. This allows SOAP intermediaries to provide value-added services. Headers, as shown later, may be inspected, inserted, deleted or forwarded by SOAP nodes encountered along a SOAP message path. (It should be kept in mind, though, that the SOAP specifications do not deal with what the contents of header elements are, or how SOAP messages are routed between nodes, or the manner by which the route is determined and so forth. These are a part of the overall application, and could be the subject of other specifications.)

The SOAP body is the mandatory element within the SOAP `env:Envelope`, which implies that this is where the main end-to-end information conveyed in a SOAP message must be carried.

A pictorial representation of the SOAP message in Example 1 is as follows.



In Example 1, the header contains two header blocks, each of which is defined in its own XML namespace and which represent some aspect pertaining to the overall processing of the body of the SOAP message. For this travel reservation application, such "meta" information pertaining to the overall request is a `reservation` header block which provides a reference and time stamp for this instance of a reservation, and the traveller's identity in the `passenger` block.

The header blocks `reservation` and `passenger` must be processed by the next SOAP intermediary encountered in the message path or, if there is no intermediary, by the ultimate recipient of the message. The fact that it is targeted at the next SOAP node encountered *en route* is indicated by the presence of the attribute `env:role` with the value "http://www.w3.org/2003/05/soap-envelope/role/next" (hereafter simply "next"), which is a role that all SOAP nodes must be willing to play. The presence of an `env:mustUnderstand` attribute with value "true" indicates that the node(s) processing the header must absolutely process these header blocks in a manner consistent with their specifications, or else not process the message at all and throw a fault. Note that whenever a header block is processed, either because it is marked `env:mustUnderstand="true"` or for another reason, the block must be processed in accordance with the specifications for that block. Such header block specifications are application defined and not a part of SOAP. Section 3 will elaborate further on SOAP message processing based on the values of these attributes.

The choice of what data is placed in a header block and what goes in the SOAP body are decisions taken at the time of application design. The main point to keep in mind is that header blocks may be targeted at various nodes that might be encountered along a message's path from a sender to the ultimate recipient. Such intermediate SOAP nodes may provide value-added services based on data in such headers. In Example 1, the passenger data is placed in a header block to illustrate the use of this data at a SOAP intermediary to do some additional processing. For example, as shown later in section 5.1, the outbound message is altered by the SOAP intermediary by having the travel policies pertaining to this passenger appended to the message as another header block.

The `env:Body` element and its associated child elements, `itinerary` and `lodging`, are intended for exchange of information between the initial SOAP sender and the SOAP node which assumes the role of the ultimate SOAP receiver in the message path, which is the travel service application. Therefore, the `env:Body` and its contents are implicitly targeted and are expected to be understood by the ultimate receiver. The means by which a SOAP node assumes such a role is not defined by the SOAP specification, and is determined as a part of the overall application semantics and associated message flow.

Note that a SOAP intermediary may decide to play the role of the ultimate SOAP receiver for a given message transfer, and thus process the `env:Body`. However, even though this sort of a behavior cannot be prevented, it is not something that should be done lightly as it may pervert the intentions of the message's sender, and have undesirable side effects (such as not processing header blocks that might be targeted at intermediaries further along the message path).

A SOAP message such as that in Example 1 may be transferred by different underlying protocols and used in a variety of message exchange patterns. For example, for a Web-based access to a travel service application, it could be placed in the body of a HTTP POST request. In another protocol binding, it might be sent in an email message (see section 4.2). Section 4 will describe how SOAP messages may be conveyed by a variety of underlying protocols. For the time being, it is assumed that a mechanism exists for message transfer and the remainder of this section concentrates on the details of the SOAP messages and their processing.

2.2 SOAP Message Exchange

SOAP Version 1.2 is a simple messaging framework for transferring information specified in the form of an XML infoset between an initial SOAP sender and an ultimate SOAP receiver. The more interesting scenarios typically involve multiple message exchanges between these two nodes. The simplest such exchange is a request-response pattern. Some early uses of [SOAP 1.1] emphasized the use of this pattern as means for conveying remote procedure calls (RPC), but it is important to note that not all SOAP request-response exchanges can or need to be modelled as RPCs. The latter is used when there is a need to model a certain programmatic behavior, with the exchanged messages conforming to a pre-defined description of the remote call and its return.

A much larger set of usage scenarios than that covered by the request-response pattern can be modeled simply as XML-based content exchanged in SOAP messages to form a back-and-forth "conversation", where the semantics are at the level of the sending and receiving applications. Section 2.2.1 covers the case of XML-based content exchanged in SOAP messages between the travel reservation application and the travel service application in a conversational pattern, while section 2.2.2 provides an example of an exchange modeled as an RPC.

2.2.1 Conversational Message Exchanges

Continuing with the travel request scenario, Example 2 shows a SOAP message returned from the travel service in response to the reservation request message in Example 1. This response seeks to refine some information in the request, namely the choice of airports in the departing city.

Example 2

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
   <m:dateAndTime>2001-11-29T13:35:00.000-05:00</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n="http://mycompany.example.com/employees"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
   <n:name>Åke Jógvan Øyvind</n:name>
  </n:passenger>
 </env:Header>
 <env:Body>
  <p:itineraryClarification
    xmlns:p="http://travelcompany.example.org/reservation/travel">
   <p:departure>
     <p:departing>
       <p:airportChoices>
          JFK LGA EWR
       </p:airportChoices>
     </p:departing>
   </p:departure>
   <p:return>
```

```
      <p:arriving>
        <p:airportChoices>
          JFK LGA EWR
        </p:airportChoices>
      </p:arriving>
    </p:return>
   </p:itineraryClarification>
  </env:Body>
 </env:Envelope>
```

SOAP message sent in response to the message in Example 1

As described earlier, the `env:Body` contains the primary content of the message, which in this example includes a list of the various alternatives for the airport, conforming to a schema definition in the XML namespace http://travelcompany.example.org/reservation/travel. In this example, the header blocks from Example 1 are returned (with some sub-element values altered) in the response. This could allow message correlation at the SOAP level, but such headers are very likely to also have other application-specific uses.

The message exchange in Examples 1 and 2 are cases where XML-based content conforming to some application-defined schema are exchanged via SOAP messages. Once again, a discussion of the means by which such messages are transferred is deferred to section 4.

It is easy enough to see how such exchanges can build up to a multiple back-and-forth "conversational" message exchange pattern. Example 3 shows a SOAP message sent by the travel reservation application in response to that in Example 2 choosing one from the list of available airports. The header block `reservation` with the same value of the `reference` sub-element accompanies each message in this conversation, thereby offering a way, should it be needed, to correlate the messages exchanged between them at the application level.

Example 3

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation
     xmlns:m="http://travelcompany.example.org/reservation"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
    <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
    <m:dateAndTime>2001-11-29T13:36:50.000-05:00</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n="http://mycompany.example.com/employees"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
   <n:name>Åke Jógvan Øyvind</n:name>
  </n:passenger>
 </env:Header>
 <env:Body>
  <p:itinerary
   xmlns:p="http://travelcompany.example.org/reservation/travel">
   <p:departure>
     <p:departing>LGA</p:departing>
   </p:departure>
   <p:return>
     <p:arriving>EWR</p:arriving>
   </p:return>
  </p:itinerary>
 </env:Body>
</env:Envelope>
```

Response to the message in Example 2 continuing a conversational message exchange

2.2.2 Remote Procedure Calls

One of the design goals of SOAP Version 1.2 is to encapsulate remote procedure call functionality using the extensibility and flexibility of XML. SOAP Part 2 section 4 has defined a uniform representation for RPC invocations and responses carried in SOAP messages. This section continues with the travel reservation scenario to illustrate the use of SOAP messages to convey remote procedure calls and their return.

To that end, the next example shows the payment for the trip using a credit card. (It is assumed that the conversational exchanges described in section 2.2.1 have resulted in a confirmed itinerary.) Here, it is further assumed that the payment happens in the context of an overall transaction where the credit card is charged only when the travel and the lodging (not shown in any example, but presumably reserved in a similar manner) are both confirmed. The travel reservation application provides credit card information and the successful completion of the different activities results in the card being charged and a reservation code returned. This reserve-and-charge interaction between the travel reservation application and the travel service application is modeled as a SOAP RPC.

To invoke a SOAP RPC, the following information is needed:

1. The address of the target SOAP node.
2. The procedure or method name.
3. The identities and values of any arguments to be passed to the procedure or method together with any output parameters and return value.
4. A clear separation of the arguments used to identify the Web resource which is the actual target for the RPC, as contrasted with those that convey data or control information used for processing the call by the target resource.
5. The message exchange pattern which will be employed to convey the RPC, together with an identification of the so-called "Web Method" (on which more later) to be used.
6. Optionally, data which may be carried as a part of SOAP header blocks.

Such information may be expressed by a variety of means, including formal Interface Definition Languages (IDL). Note that SOAP does not provide any IDL, formal or informal. Note also that the above information differs in subtle ways from information generally needed to invoke other, non-SOAP RPCs.

Regarding Item 1 above, there is, from a SOAP perspective, a SOAP node which "contains" or "supports" the target of the RPC. It is the SOAP node which (appropriately) adopts the role of the ultimate SOAP receiver. As required by Item 1, the ultimate recipient can identify the target of the named procedure or method by looking for its URI. The manner in which the target URI is made available depends on the underlying protocol binding. One possibility is that the URI identifying the target is carried in a SOAP header block. Some protocol bindings, such as the SOAP HTTP binding defined in [SOAP Part2], offer a mechanism for carrying the URI outside the SOAP message. In general, one of the properties of a protocol binding specification must be a description of how the target URI is carried as a part of the binding. Section 4.1 provides some concrete examples of how the URI is carried in the case of the standardized SOAP protocol binding to HTTP.

Item 4 and Item 5 above are required to ensure that RPC applications that employ SOAP can do so in a manner which is compatible with the architectural principles of the World Wide Web. Section 4.1.3 discusses how the information provided by items 4 and 5 are utilized.

For the remainder of this section, it is assumed that the RPC conveyed in a SOAP message as shown in Example 4 is appropriately targeted and dispatched. The purpose of this section is to highlight the syntactical aspects of RPC requests and returns carried within a SOAP message.

Example 4

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
 <env:Header>
   <t:transaction
           xmlns:t="http://thirdparty.example.org/transaction"
           env:encodingStyle="http://example.com/encoding"
           env:mustUnderstand="true" >5</t:transaction>
 </env:Header>
 <env:Body>
  <m:chargeReservation
      env:encodingStyle="http://www.w3.org/2003/05/soap-encoding"
         xmlns:m="http://travelcompany.example.org/">
   <m:reservation xmlns:m="http://travelcompany.example.org/reservation">
    <m:code>FT35ZBQ</m:code>
   </m:reservation>
   <o:creditCard xmlns:o="http://mycompany.example.com/financial">
    <n:name xmlns:n="http://mycompany.example.com/employees">
           Åke Jógvan Øyvind
```

```
    </n:name>
    <o:number>123456789099999</o:number>
    <o:expiration>2005-02</o:expiration>
   </o:creditCard>
  </m:chargeReservation>
 </env:Body>
</env:Envelope>
```

SOAP RPC request with a mandatory header and two input (or "in") parameters

The RPC itself is carried as a child of the `env:Body` element, and is modelled as a `struct` which takes the name of the procedure or method, in this case `chargeReservation`. (A `struct` is a concept from the SOAP Data Model defined in [SOAP Part2] that models a structure or record type that occurs in some common programming languages.) The design of the RPC in the example (whose formal description has not been explicitly provided) takes two input (or "in") parameters, the `reservation` corresponding to the planned trip identified by the reservation `code`, and the `creditCard` information. The latter is also a `struct`, which takes three elements, the card holder's `name`, the card `number` and an `expiration` date.

In this example, the `env:encodingStyle` attribute with the value http://www.w3.org/2003/05/soap-encoding shows that the contents of the `chargeReservation` structure have been serialized according to the SOAP encoding rules, i.e., the particular rules defined in SOAP Part 2 section 3. Even though SOAP specifies this particular encoding scheme, its use is optional and the specification makes clear that other encoding schemes may be used for application-specific data within a SOAP message. It is for this purpose that it provides the `env:encodingStyle` attribute to qualify header blocks and body sub-elements. The choice of the value for this attribute is an application-specific decision and the ability of a caller and callee to interoperate is assumed to have been settled "out-of-band". Section 5.2 shows an example of using another encoding scheme.

As noted in Item 6 above, RPCs may also require additional information to be carried, which can be important for the processing of the call in a distributed environment, but which are not a part of the formal procedure or method description. (Note, however, that providing such additional contextual information is not specific to RPCs, but may be required in general for the processing of any distributed application.) In the example, the RPC is carried out in the context of an overall transaction which involves several activities which must all complete successfully before the RPC returns successfully. Example 4 shows how a header block `transaction` directed at the ultimate recipient (implied by the absence of the `env:role` attribute) is used to carry such information. (The value "5" is some transaction identifier set by and meaningful to the application. No further elaboration of the application-specific semantics of this header are provided here, as it is not germane to the discussion of the syntactical aspects of SOAP RPC messages.)

Let us assume that the RPC in the charging example has been designed to have the procedure description which indicates that there are two output (or "out") parameters, one providing the reference code for the reservation and the other a URL where the details of the reservation may be viewed. The RPC response is returned in the `env:Body` element of a SOAP message, which is modeled as a `struct` taking the procedure name `chargeReservation` and, as a convention, the word "Response" appended. The two output (or "out") parameters accompanying the response are the alphanumeric `code` identifying the reservation in question, and a URI for the location, `viewAt`, from where the reservation may be retrieved.

This is shown in Example 5a, where the header again identifies the transaction within which this RPC is performed.

Example 5a

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
 <env:Header>
     <t:transaction
        xmlns:t="http://thirdparty.example.org/transaction"
          env:encodingStyle="http://example.com/encoding"
           env:mustUnderstand="true">5</t:transaction>
 </env:Header>
 <env:Body>
     <m:chargeReservationResponse
         env:encodingStyle="http://www.w3.org/2003/05/soap-encoding"
             xmlns:m="http://travelcompany.example.org/">
       <m:code>FT35ZBQ</m:code>
       <m:viewAt>
         http://travelcompany.example.org/reservations?code=FT35ZBQ
```

```
        </m:viewAt>
      </m:chargeReservationResponse>
  </env:Body>
 </env:Envelope>
```

RPC response with two output (or "out") parameters for the call shown in Example 4

RPCs often have descriptions where a particular output parameter is distinguished, the so-called "return" value. The SOAP RPC convention offers a way to distinguish this "return" value from the other output parameters in the procedure description. To show this, the charging example is modified to have an RPC description that is almost the same as that for Example 5a, i.e, with the same two "out" parameters, but in addition it also has a "return" value, which is an enumeration with potential values of "confirmed" and "pending". The RPC response conforming to this description is shown in Example 5b, where the SOAP header, as before, identifies the transaction within which this RPC is performed.

Example 5b

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
 <env:Header>
    <t:transaction
       xmlns:t="http://thirdparty.example.org/transaction"
         env:encodingStyle="http://example.com/encoding"
          env:mustUnderstand="true">5</t:transaction>
 </env:Header>
 <env:Body>
    <m:chargeReservationResponse
       env:encodingStyle="http://www.w3.org/2003/05/soap-encoding"
           xmlns:rpc="http://www.w3.org/2003/05/soap-rpc"
             xmlns:m="http://travelcompany.example.org/">
       <rpc:result>m:status</rpc:result>
       <m:status>confirmed</m:status>
       <m:code>FT35ZBQ</m:code>
       <m:viewAt>
        http://travelcompany.example.org/reservations?code=FT35ZBQ
       </m:viewAt>
    </m:chargeReservationResponse>
 </env:Body>
</env:Envelope>
```

RPC response with a "return" value and two "out" parameters for the call shown in Example 4

In Example 5b, the return value is identified by the element `rpc:result`, and contains the XML Qualified Name (of type `xs:QName`) of another element within the `struct` which is `m:status`. This, in turn, contains the actual return value, "confirmed". This technique allows the actual return value to be strongly typed according to some schema. If the `rpc:result` element is absent, as is the case in Example 5a, the return value is not present or is of the type `void`.

While, in principle, using SOAP for RPC is independent of the decision to use a particular means for transferring the RPC call and its return, certain protocol bindings that support the SOAP Request-Response message exchange pattern may be more naturally suited for such purposes. A protocol binding supporting this message exchange pattern can provide the correlation between a request and a response. Of course, the designer of an RPC-based application could choose to put a correlation ID relating a call and its return in a SOAP header, thereby making the RPC independent of any underlying transfer mechanism. In any case, application designers have to be aware of all the characteristics of the particular protocols chosen for transferring SOAP RPCs, such as latency, synchrony, etc.

In the commonly used case, standardized in SOAP Part 2 section 7, of using HTTP as the underlying transfer protocol, an RPC invocation maps naturally to the HTTP request and an RPC response maps to the HTTP response. Section 4.1 provides examples of carrying RPCs using the HTTP binding.

However, it is worth keeping in mind that even though most examples of SOAP for RPC use the HTTP protocol binding, it is not limited to that means alone.

2.3 Fault Scenarios

SOAP provides a model for handling situations when faults arise in the processing of a message. SOAP

distinguishes between the conditions that result in a fault, and the ability to signal that fault to the originator of the faulty message or another node. The ability to signal the fault depends on the message transfer mechanism used, and one aspect of the binding specification of SOAP onto an underlying protocol is to specify how faults are signalled, if at all. The remainder of this section assumes that a transfer mechanism is available for signalling faults generated while processing received messages, and concentrates on the structure of the SOAP fault message.

The SOAP `env:Body` element has another distinguished role in that it is the place where such fault information is placed. The SOAP fault model (see SOAP Part 1, section 2.6) requires that all SOAP-specific and application-specific faults be reported using a *single* distinguished element, `env:Fault`, carried within the `env:Body` element. The `env:Fault` element contains two mandatory sub-elements, `env:Code` and `env:Reason`, and (optionally) application-specific information in the `env:Detail` sub-element. Another optional sub-element, `env:Node`, identifies via a URI the SOAP node which generated the fault, its absence implying that it was the ultimate recipient of the message which did so. There is yet another optional sub-element, `env:Role`, which identifies the role being played by the node which generated the fault.

The `env:Code` sub-element of `env:Fault` is itself made up of a mandatory `env:Value` sub-element, whose content is specified in the SOAP specification (see SOAP Part 1 section 5.4.6) as well as an optional `env:Subcode` sub-element.

Example 6a shows a SOAP message returned in response to the RPC request in Example 4, and indicating a failure to process the RPC.

Example 6a

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
              xmlns:rpc='http://www.w3.org/2003/05/soap-rpc'>
  <env:Body>
   <env:Fault>
     <env:Code>
       <env:Value>env:Sender</env:Value>
       <env:Subcode>
        <env:Value>rpc:BadArguments</env:Value>
       </env:Subcode>
     </env:Code>
     <env:Reason>
      <env:Text xml:lang="en-US">Processing error</env:Text>
      <env:Text xml:lang="cs">Chyba zpracování</env:Text>
     </env:Reason>
     <env:Detail>
      <e:myFaultDetails
        xmlns:e="http://travelcompany.example.org/faults">
        <e:message>Name does not match card number</e:message>
        <e:errorcode>999</e:errorcode>
      </e:myFaultDetails>
     </env:Detail>
   </env:Fault>
 </env:Body>
</env:Envelope>
```

Sample SOAP message indicating failure to process the RPC in Example 4

In Example 6a, the top-level `env:Value` uses a standardized XML Qualified Name (of type `xs:QName`) to identify that it is an `env:Sender` fault, which indicates that it is related to some syntactical error or inappropriate information in the message. (When a `env:Sender` fault is received by the sender, it is expected that some corrective action is taken before a similar message is sent again.) The `env:Subcode` element is optional, and, if present, as it is in this example, qualifies the parent value further. In Example 6a, the `env:Subcode` denotes that an RPC specific fault, `rpc:BadArguments`, defined in SOAP Part 2 section 4.4, is the cause of the failure to process the request.

The structure of the `env:Subcode` element has been chosen to be hierarchical - each child `env:Subcode` element has a mandatory `env:Value` and an optional `env:Subcode` sub-element - to allow application-specific codes to be carried. This hierarchical structure of the `env:Code` element allows for an uniform mechanism for conveying multiple level of fault codes. The top-level `env:Value` is a base fault that is specified in the SOAP Version 1.2 specifications (see SOAP Part 1 section 5.4.6) and must be understood by all SOAP nodes. Nested `env:Value`s are application-specific, and represent further elaboration or refinement of the base fault

from an application perspective. Some of these values may well be standardized, such as the RPC codes standardized in SOAP 1.2 (see SOAP Part 2 section 4.4), or in some other standards that use SOAP as an encapsulation protocol. The only requirement for defining such application-specific subcode values is that they be namespace qualified using any namespace other than the SOAP `env` namespace which defines the main classifications for SOAP faults. There is no requirement from a SOAP perspective that applications need to understand, or even look at all levels of the subcode values.

The `env:Reason` sub-element is not meant for algorithmic processing, but rather for human understanding; so, even though this is a mandatory item, the chosen value need not be standardized. Therefore all that is required is that it reasonably accurately describe the fault situation. It must have one or more `env:Text` sub-elements, each with a unique `xml:lang` attribute, which allows applications to make the fault reason available in multiple languages. (Applications could negotiate the language of the fault text using a mechanism built using SOAP headers; however this is outside the scope of the SOAP specifications.)

The absence of a `env:Node` sub-element within `env:Fault` in Example 6a implies that it is generated by the ultimate receiver of the call. The contents of `env:Detail`, as shown in the example, are application-specific.

During the processing of a SOAP message, a fault may also be generated if a mandatory header element is not understood or the information contained in it cannot be processed. Errors in processing a header block are also signalled using a `env:Fault` element within the `env:Body`, but with a particular distinguished header block, `env:NotUnderstood`, that identifies the offending header block.

Example 6b shows an example of a response to the RPC in Example 4 indicating a failure to process the `t:transaction` header block. Note the presence of the `env:MustUnderstand` fault code in the `env:Body`, and the identification of the header not understood using an (unqualified) attribute, `qname`, in the special (empty) header block `env:NotUnderstood`.

Example 6b

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope>
 <env:Header>
    <env:NotUnderstood qname="t:transaction"
                xmlns:t="http://thirdparty.example.org/transaction"/>
 </env:Header>
 <env:Body>
  <env:Fault>
   <env:Code>
    <env:Value>env:MustUnderstand</env:Value>
   </env:Code>
   <env:Reason>
      <env:Text xml:lang="en-US">Header not understood</env:Text>
      <env:Text xml:lang="fr">En-tête non compris</env:Text>
   </env:Reason>
  </env:Fault>
 </env:Body>
</env:Envelope>
```

Sample SOAP message indicating failure to process the header block in Example 4

If there were several mandatory header blocks that were not understood, then each could be identified by its `qname` attribute in a series of such `env:NotUnderstood` header blocks.

3. SOAP Processing Model

Having established the various syntactical aspects of a SOAP message as well as some basic message exchange patterns, this section provides a general overview of the SOAP processing model (specified in SOAP Part 1, section 2). The SOAP processing model describes the actions taken by a SOAP node on receiving a SOAP message.

Example 7a shows a SOAP message with several header blocks (with their contents omitted for brevity). Variations of this will be used in the remainder of this section to illustrate various aspects of the processing model.

Example 7a

```
<?xml version="1.0" ?>
 <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
   <env:Header>
     <p:oneBlock xmlns:p="http://example.com"
            env:role="http://example.com/Log">
     ...
     ...
     </p:oneBlock>
     <q:anotherBlock xmlns:q="http://example.com"
       env:role="http://www.w3.org/2003/05/soap-envelope/role/next">
     ...
     ...
     </q:anotherBlock>
     <r:aThirdBlock xmlns:r="http://example.com">
     ...
     ...
     </r:aThirdBlock>
   </env:Header>
   <env:Body >
     ...
     ...
   </env:Body>
 </env:Envelope>
```

SOAP message showing a variety of header blocks

The SOAP processing model describes the (logical) actions taken by a SOAP node on receiving a SOAP message. There is a requirement for the node to analyze those parts of a message that are SOAP-specific, namely those elements in the SOAP "env" namespace. Such elements are the envelope itself, the header element and the body element. A first step is, of course, the overall check that the SOAP message is syntactically correct. That is, it conforms to the SOAP XML infoset subject to the restrictions on the use of certain XML constructs - Processing Instructions and Document Type Definitions - as defined in SOAP Part 1, section 5.

3.1 The "role" Attribute

Further processing of header blocks and the body depend on the role(s) assumed by the SOAP node for the processing of a given message. SOAP defines the (optional) `env:role` attribute - syntactically, `xs:anyURI` - that may be present in a header block, which identifies the role played by the intended target of that header block. A SOAP node is required to process a header block if it assumes the role identified by the value of the URI. How a SOAP node assumes a particular role is not a part of the SOAP specifications.

Three standardized roles have been defined (see SOAP Part 1, section 2.2), which are

- "http://www.w3.org/2003/05/soap-envelope/role/none" (hereafter simply "none")
- "http://www.w3.org/2003/05/soap-envelope/role/next" (hereafter simply "next"), and
- "http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver" (hereafter simply "ultimateReceiver").

In Example 7a, the header block `oneBlock` is targeted at any SOAP node that plays the application-defined role defined by the URI http://example.com/Log. For purposes of illustration, it is assumed that the specification for such a header block requires that any SOAP node adopting this role log the entire message.

Every SOAP node receiving a message with a header block that has a `env:role` attribute of "next" must be capable of processing the contents of the element, as this is a standardized role that every SOAP node must be willing to assume. A header block thus attributed is one which is expected to be examined and (possibly) processed by the next SOAP node along the path of a message, assuming that such a header has not been removed as a result of processing at some node earlier in the message path.

In Example 7a, the header block `anotherBlock` is targeted at the next node in the message path. In this case, the SOAP message received by the node playing the application-defined role of "http://example.com /Log", must also be willing to play the SOAP-defined role of "next". This is also true for the node which is the ultimate recipient of the message, as it obviously (and implicitly) also plays the "next" role by virtue of being next in the message path.

The third header block, `aThirdBlock`, in Example 7a does not have the `env:role` attribute. It is targeted at a SOAP node which assumes the "ultimateReceiver" role. The "ultimateReceiver" role (which can be explicitly

declared or is implicit if the `env:role` attribute is absent in a header block) is played by a SOAP node that assumes the role of the ultimate recipient of a particular SOAP message. The absence of a `env:role` attribute in the `aThirdBlock` header block means that this header element is targeted at the SOAP node that assumes the "ultimateReceiver" role.

Note that the `env:Body` element does not have a `env:role` attribute. The body element is *always* targeted at the SOAP node that assumes the "ultimateReceiver" role. In that sense, the body element is just like a header block targeted at the ultimate receiver, but it has been distinguished to allow for SOAP nodes (typically SOAP intermediaries) to skip over it if they assume roles other than that of the ultimate receiver. SOAP does not prescribe any structure for the `env:Body` element, except that it recommends that any sub-elements be XML namespace qualified. Some applications, such as that in Example 1, may choose to organize the sub-elements of `env:Body` in blocks, but this is not of concern to the SOAP processing model.

The other distinguished role for the `env:Body` element, as the container where information on SOAP-specific faults, i.e., failure to process elements of a SOAP message, is placed has been described previously in section 2.3.

If a header element has the standardized `env:role` attribute with value "none", it means that no SOAP node should process the contents, although a node may need to examine it if the content are data referenced by another header element that is targeted at the particular SOAP node.

If the `env:role` attribute has an empty value, i.e., `env:role=""`, it means that the relative URI identifying the role is resolved to the base URI for the SOAP message in question. SOAP Version 1.2 does not define a base URI for a SOAP message, but defers to the mechanisms defined in [XMLBase] for deriving the base URI, which can be used to make any relative URIs absolute. One such mechanism is for the protocol binding to establish a base URI, possibly by reference to the encapsulating protocol in which the SOAP message is embedded for transport. (In fact, when SOAP messages are transported using HTTP, SOAP Part 2 section 7.1.2 defines the base URI as the Request-URI of the HTTP request, or the value of the HTTP Content-Location header.)

The following table summarizes the applicable standardized roles that may be assumed at various SOAP nodes. ("Yes" and "No" means that the corresponding node does or does not, respectively, play the named role.)

Role	absent	"none"	"next"	"ultimateReceiver"
Node				
initial sender	not applicable	not applicable	not applicable	not applicable
intermediary	no	no	yes	no
ultimate receiver	yes	no	yes	yes

3.2 The "mustUnderstand" Attribute

Example 7b augments the previous example by introducing another (optional) attribute for header blocks, the `env:mustUnderstand` attribute.

Example 7b

```
<?xml version="1.0" ?>
 <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
   <env:Header>
     <p:oneBlock xmlns:p="http://example.com"
           env:role="http://example.com/Log"
               env:mustUnderstand="true">
     ...
     ...
     </p:oneBlock>
     <q:anotherBlock xmlns:q="http://example.com"
       env:role="http://www.w3.org/2003/05/soap-envelope/role/next">
     ...
     ...
     </q:anotherBlock>
     <r:aThirdBlock xmlns:r="http://example.com">
     ...
     ...
```

```
      </r:aThirdBlock>
    </env:Header>
    <env:Body >
      ...
      ...
    </env:Body>
  </env:Envelope>
```

SOAP message showing a variety of header blocks, one of which is mandatory for processing

After a SOAP node has correctly identified the header blocks (and possibly the body) targeted at itself using the `env:role` attribute, the additional attribute, `env:mustUnderstand`, in the header elements determines further processing actions that have to be taken. In order to ensure that SOAP nodes do not ignore header blocks which are important to the overall purpose of the application, SOAP header blocks also provide for the additional optional attribute, `env:mustUnderstand`, which, if "true", means that the targeted SOAP node **must** process the block according to the specification of that block. Such a block is colloquially referred to as a mandatory header block. In fact, processing of the SOAP message must not even start until the node has identified all the mandatory header blocks targeted at itself, and "understood" them. Understanding a header means that the node must be prepared to do whatever is described in the specification of that block. (Keep in mind that the specifications of header blocks are not a part of the SOAP specifications.)

In Example 7b, the header block `oneBlock` is marked with a `env:mustUnderstand` value set to "true", which means that it is mandatory to process this block if the SOAP node plays the role identified by "http://example.com/Log". The other two header blocks are not so marked, which means that SOAP node at which these blocks are targeted need not process them. (Presumably the specifications for these blocks allow for this.)

A `env:mustUnderstand` value of "true" means that the SOAP node must process the header with the semantics described in that header's specification, or else generate a SOAP fault. Processing the header appropriately may include removing the header from any generated SOAP message, reinserting the header with the same or altered value, or inserting a new header. The inability to process a mandatory header requires that all further processing of the SOAP message cease, and a SOAP fault be generated. The message is not forwarded any further.

The `env:Body` element has no `env:mustUnderstand` attribute but it *must* be processed by the ultimate recipient. In Example 7b, the ultimate recipient of the message - the SOAP node which plays the "ultimateReceiver" role - must process the `env:Body` and may process the header block `aThirdBlock`. It may also process the header block `anotherBlock`, as it is targeted at it (in the role of "next") but it is not mandatory to do so if the specifications for processing the blocks do not demand it. (If the specification for `anotherBlock` demanded that it must be processed at the next recipient, it would have required that it be marked with a `env:mustUnderstand`="true".)

The role(s) a SOAP node plays when processing a SOAP message can be determined by many factors. The role could be known *a priori*, or set by some out-of-band means, or a node can inspect all parts of a received message to determine which roles it will assume before processing the message. An interesting case arises when a SOAP node, during the course of processing a message, decides that there are additional roles that it needs to adopt. No matter when this determination is made, externally it must appear as though the processing model has been adhered to. That is, it must appear as though the role had been known from the start of the processing of the message. In particular, the external appearance must be that the `env:mustUnderstand` checking of any headers with those additional roles assumed was performed before any processing began. Also, if a SOAP node assumes such additional roles, it must ensure that it is prepared to do everything that the specifications for those roles require.

The following table summarizes how the processing actions for a header block are qualified by the `env:mustUnderstand` attribute with respect to a node that has been appropriately targeted (via the `env:role` attribute).

Node	intermediary	ultimate receiver
mustUnderstand		
"true"	must process	must process
"false"	may process	may process
absent	may process	may process

As a result of processing a SOAP message, a SOAP node may generate a single SOAP fault if it fails to process a message, or, depending on the application, generate additional SOAP messages for consumption at other SOAP nodes. SOAP Part 1 section 5.4 describes the structure of the fault message while the SOAP processing model defines the conditions under which it is generated. As illustrated previously in section 2.3, a SOAP fault is a SOAP message with a standardized `env:Body` sub-element named `env:Fault`.

SOAP makes a distinction between generating a fault and ensuring that the fault is returned to the originator of the message or to another appropriate node which can benefit from this information. However, whether a generated fault can be propagated appropriately depends on the underlying protocol binding chosen for the SOAP message message exchange. The specification does not define what happens if faults are generated during the propagation of one-way messages. The only normative underlying protocol binding, which is the SOAP HTTP binding, offers the HTTP response as a means for reporting a fault in the incoming SOAP message. (See Section 4 for more details on SOAP protocol bindings.)

3.3 The "relay" Attribute

SOAP Version 1.2 defines another optional attribute for header blocks, `env:relay` of type `xs:boolean`, which indicates if a header block targeted at a SOAP intermediary must be relayed if it is *not* processed.

Note that if a header block *is* processed, the SOAP processing rules (see SOAP Part 1 section 2.7.2) requires that it be removed from the outbound message. (It may, however, be reinserted, either unchanged or with its contents altered, if the processing of other header blocks determines that the header block be retained in the forwarded message.) The default behavior for *an un*processed header block targeted at a role played by a SOAP intermediary is that it must be removed before the message is relayed.

The reason for this choice of default is to lean on the side of safety by ensuring that a SOAP intermediary make no assumptions about the survivability past itself of a header block targeted at a role it assumes, and representing some value-added feature, particularly if it chooses not to process the header block, very likely because it does not "understand" it. That is because certain header blocks represent hop-by-hop features, and it may not make sense to unknowingly propagate it end-to-end. As an intermediary may not be in a position to make this determination, it was thought that it would be safer if unprocessed header blocks were removed before the message was relayed.

However, there are instances when an application designer would like to introduce a new feature, manifested through a SOAP header block, targeted at *any* capable intermediary which might be encountered in the SOAP message path. Such a header block would be available to those intermediaries that "understood" it, but ignored and relayed onwards by those that did not. Being a new feature, the processing software for this header block may be implemented, at least initially, in some but not all SOAP nodes. Marking such a header block with `env:mustUnderstand` = "false" is obviously needed, so that intermediaries that have not implemented the feature do not generate a fault. To circumvent the default rule of the processing model, marking a header block with the additional attribute `env:relay` with the value "true" allows the intermediary to forward the header block targeted at itself in the event that it chooses not to process it.

Targeting the header block at the role "next" together with the `env:relay` attribute set to "true" can always serve to ensure that each intermediary has a chance to examine the header, because one of the anticipated uses of the "next" role is with header blocks that carry information that are expected to persist along a SOAP message path. Of course, the application designer can always define a custom role that allows targetting at specific intermediaries that assume this role. Therefore, there is no restriction on the use of the `env:relay` attribute with any role except of course the roles of "none" and "ultimateReceiver", for which it is meaningless.

Example 7c shows the use of the `env:relay` attribute.

Example 7c

```
<?xml version="1.0" ?>
 <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
   <env:Header>
     <p:oneBlock xmlns:p="http://example.com"
           env:role="http://example.com/Log"
               env:mustUnderstand="true">
     ...
```

```
      ...
      </p:oneBlock>
      <q:anotherBlock xmlns:q="http://example.com"
        env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
        env:relay="true">
      ...
      ...
      </q:anotherBlock>
      <r:aThirdBlock xmlns:r="http://example.com">
      ...
      ...
      </r:aThirdBlock>
    </env:Header>
    <env:Body >
      ...
      ...
    </env:Body>
  </env:Envelope>
```

SOAP message showing a variety of header blocks, one of which must be relayed if unprocessed.

The header block `q:anotherBlock`, targeted at the "next" node in the message path, has the additional attribute `env:relay`="true". A SOAP node receiving this message may process this header block if it "understands" it, but if it does so the processing rules require that this header block be removed before forwarding. However, if the SOAP node chooses to ignore this header block, which it can because it is not mandatory to process it, as indicated by the absence of the `env:mustUnderstand` attribute, then it must forward it.

Processing the header block `p:oneBlock` is mandatory and the SOAP processing rules require that it not be relayed, unless the processing of some other header block requires that it be present in the outbound message. The header block `r:aThirdBlock` does not have an `env:relay` attribute, which is equivalent to having it with the value `env:relay`="false". Hence, this header is not forwarded if it is not processed.

SOAP 1.2 Part 1 Table 3 summarizes the conditions which determine when a SOAP intermediary assuming a given role is allowed to forward unprocessed header blocks.

4. Using Various Protocol Bindings

SOAP messages may be exchanged using a variety of "underlying" protocols, including other application layer protocols. The specification of how SOAP messages may be passed from one SOAP node to another using an underlying protocol is called a SOAP binding. [SOAP Part1] defines a SOAP message in the form of an [XML Infoset], i.e., in terms of element and attribute information items of an abstract "document" called the `env:Envelope` (see SOAP Part 1, section 5). Any SOAP `env:Envelope` infoset representation will be made concrete through a protocol binding, whose task, among other things, it is to provide a serialized representation of the infoset that can be conveyed to the next SOAP node in the message path in a manner such that the infoset can be reconstructed without loss of information. In typical examples of SOAP messages, and certainly in all the examples in this primer, the serialization shown is that of a well-formed [XML 1.0] document. However, there may be other protocol bindings - for example a protocol binding between two SOAP nodes over a limited bandwidth interface - where an alternative, compressed serialization of the same infoset may be chosen. Another binding, chosen for a different purpose, may provide a serialization which is an encrypted structure representing the same infoset.

In addition to providing a concrete realization of a SOAP infoset between adjacent SOAP nodes along a SOAP message path, a protocol binding provides the mechanisms to support features that are needed by a SOAP application. A feature is a specification of a certain functionality provided by a binding. A feature description is identified by a URI, so that all applications referencing it are assured of the same semantics. For example, a typical usage scenario might require many concurrent request-response exchanges between adjacent SOAP nodes, in which case the feature that is required is the ability to correlate a request with a response. Other examples includes "an encrypted channel" feature, or a "reliable delivery channel" feature, or a particular SOAP message exchange pattern feature.

A SOAP binding specification (see SOAP Part 1 section 4) describes, among other things, which (if any) features it provides. Some features may be provided natively by the underlying protocol. If the feature is not available through the binding, it may be implemented within the SOAP envelope, using SOAP header blocks. The specification of a feature implemented using SOAP header blocks is called a SOAP module.

For example, if SOAP message exchanges were being transported directly over a datagram protocol like UDP, obviously the message correlation feature would have to be provided elsewhere, either directly by the application or more likely as a part of the SOAP infosets being exchanged. In the latter case, the message correlation feature has a binding-specific expression within the SOAP envelope, i.e., as a SOAP header block, defined in a "Request-Response Correlation" module identified by a URI. However, if the SOAP infosets were being exchanged using an underlying protocol that was itself request/response, the application could implicitly "inherit" this feature provided by the binding, and no further support need be provided at the application or the SOAP level. (In fact, the HTTP binding for SOAP takes advantage of just this feature of HTTP.)

However, a SOAP message may travel over several hops between a sender and the ultimate receiver, where each hop may be a different protocol binding. In other words, a feature (e.g., message correlation, reliability etc.) that is supported by the protocol binding in one hop may not be supported by another along the message path. SOAP itself does not provide any mechanism for hiding the differences in features provided by different underlying protocols. However, any feature that is required by a particular application, but which may not be available in the underlying infrastructure along the *anticipated* message path, can be compensated for by being carried as a part of the SOAP message infoset, i.e., as a SOAP header block specified in some module.

Thus it is apparent that there are a number of issues that have to be tackled by an application designer to accomplish particular application semantics, including how to take advantage of the native features of underlying protocols that are available for use in the chosen environment. SOAP Part 1 section 4.2 provides a general framework for describing how SOAP-based applications may choose to use the features provided by an underlying protocol binding to accomplish particular application semantics. It is intended to provide guidelines for writing interoperable protocol binding specifications for exchanging SOAP messages.

Among other things, a binding specification must define one particular feature, namely the message exchange pattern(s) that it supports. [SOAP Part2] defines two such message exchange patterns, namely a SOAP Request-Response message exchange pattern where one SOAP message is exchanged in each direction between two adjacent SOAP nodes, and a SOAP Response message exchange pattern which consists of a non-SOAP message acting as a request followed by a SOAP message included as a part of the response.

[SOAP Part2] also offers the application designer a general feature called the SOAP Web Method feature that allows applications full control over the choice of the so-called "Web method" - one of GET, POST, PUT, DELETE whose semantics are as defined in the [HTTP 1.1] specifications - that may be used over the binding. This feature is defined to ensure that applications using SOAP can do so in a manner which is compatible with the architectural principles of the World Wide Web. (Very briefly, the simplicity and scalability of the Web is largely due to the fact that there are a few "generic" methods (GET, POST, PUT, DELETE) which can be used to interact with any resource made available on the Web via a URI.) The SOAP Web Method feature is supported by the SOAP HTTP binding, although, in principle, it is available to all SOAP underlying protocol bindings.

SOAP Part 2 section 7 specifies one standardized protocol binding using the binding framework of [SOAP Part1], namely how SOAP is used in conjunction with HTTP as the underlying protocol. SOAP Version 1.2 restricts itself to the definition of a HTTP binding allowing only the use of the POST method in conjunction with the Request-Response message exchange pattern and the GET method with the SOAP Response message exchange pattern. Other specifications in future could define SOAP bindings to HTTP or other transports that make use of the other Web methods (i.e., PUT, DELETE).

The next sections show examples of two underlying protocol bindings for SOAP, namely those to [HTTP 1.1] and email. It should be emphasized again that the only normative binding for SOAP 1.2 messages is to [HTTP 1.1]. The examples in section 4.2 showing email as a transport mechanism for SOAP is simply meant to suggest that other choices for the transfer of SOAP messages are possible, although not standardized at this time. A W3C Note [SOAP Email Binding] offers an application of the SOAP protocol binding framework of [SOAP Part1] by describing an experimental binding of SOAP to email transport, specifically [RFC 2822]-based message transport.

4.1 The SOAP HTTP Binding

HTTP has a well-known connection model and a message exchange pattern. The client identifies the server

via a URI, connects to it using the underlying TCP/IP network, issues a HTTP request message and receives a HTTP response message over the same TCP connection. HTTP implicitly correlates its request message with its response message; therefore, an application using this binding can chose to infer a correlation between a SOAP message sent in the body of a HTTP request message and a SOAP message returned in the HTTP response. Similarly, HTTP identifies the server endpoint via a URI, the Request-URI, which can also serve as the identification of a SOAP node at the server.

HTTP allows for multiple intermediaries between the initial client and the origin server identified by the Request-URI, in which case the request/response model is a series of such pairs. Note, however, that HTTP intermediaries are distinct from SOAP intermediaries.

The HTTP binding in [SOAP Part2] makes use of the SOAP Web Method feature to allow applications to choose the so-called Web method - restricting it to one of GET or POST - to use over the HTTP message exchange. In addition, it makes use of two message exchange patterns that offer applications two ways of exchanging SOAP messages via HTTP: 1) the use of the HTTP POST method for conveying SOAP messages in the bodies of HTTP request and response messages, and 2) the use of the HTTP GET method in a HTTP request to return a SOAP message in the body of a HTTP response. The first usage pattern is the HTTP-specific instantiation of a binding feature called the SOAP request-response message exchange pattern, while the second uses a feature called the SOAP response message exchange pattern.

The purpose of providing these two types of usages is to accommodate the two interaction paradigms which are well established on the World Wide Web. The first type of interaction allows for the use of data within the body of a HTTP POST to create or modify the state of a resource identified by the URI to which the HTTP request is destined. The second type of interaction pattern offers the ability to use a HTTP GET request to obtain a representation of a resource without altering its state in any way. In the first case, the SOAP-specific aspect of concern is that the body of the HTTP POST request is a SOAP message which has to be processed (per the SOAP processing model) as a part of the application-specific processing required to conform to the POST semantics. In the second case, the typical usage that is forseen is the case where the representation of the resource that is being requested is returned not as a HTML, or indeed a generic XML document, but as a SOAP message. That is, the HTTP content type header of the response message identifies it as being of media type "application/soap+xml". Presumably, there will be publishers of resources on the Web who determine that such resources are best retrieved and made available in the form of SOAP messages. Note, however, that resources can, in general, be made available in multiple representations, and the desired or preferred representation is indicated by the requesting application using the HTTP Accept header.

One further aspect of the SOAP HTTP binding is the question of how an application determines which of these two types of message exchange patterns to use. [SOAP Part2] offers guidance on circumstances when applications may use one of the two specified message exchange patterns. (It is guidance - albeit a strong one - as it is phrased in the form of a "SHOULD" in the specifications rather than an absolute requirement identified by the word "MUST", where these words are interpreted as defined in the IETF [RFC 2119].) The SOAP response message exchange pattern with the HTTP GET method is used when an application is assured that the message exchange is for the purposes of information retrieval, where the information resource is "untouched" as a result of the interaction. Such interactions are referred to as safe and idempotent in the HTTP specification. As the HTTP SOAP GET usage does not allow for a SOAP message in the request, applications that need features in the outbound interaction that can only be supported by a binding-specific expression within the SOAP infoset (i.e., as SOAP header blocks) obviously cannot make use of this message exchange pattern. Note that the HTTP POST binding is available for use in all cases.

The following subsections provide examples of the use of these two message exchange patterns defined for the HTTP binding.

4.1.1. SOAP HTTP GET Usage

Using the HTTP binding with the SOAP Response message exchange pattern is restricted to the HTTP GET method. This means that the response to a HTTP GET request from a requesting SOAP node is a SOAP message in the HTTP response.

Example 8a shows a HTTP GET directed by the traveller's application (in the continuing travel reservation scenario) at the URI `http://travelcompany.example.org/reservations?code=FT35ZBQ` where the traveler's itinerary may be viewed. (How this URL was made available can be seen in Example 5a.)

Example 8a

```
GET /travelcompany.example.org/reservations?code=FT35ZBQ  HTTP/1.1
Host: travelcompany.example.org
Accept: text/html;q=0.5, application/soap+xml
```

HTTP GET Request

The HTTP Accept header is used to indicate the preferred representation of the resource being requested, which in this example is an "application/soap+xml" media type for consumption by a machine client, rather than the "text/html" media type for rendition by a browser client for consumption by a human.

Example 8b shows the HTTP response to the GET in Example 8a. The body of the HTTP response contains a SOAP message showing the travel details. A discussion of the contents of the SOAP message is postponed until section 5.2 , as it is not relevant, at this point, to understanding the HTTP GET binding usage.

Example 8b

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
       env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
   <m:dateAndTime>2001-11-30T16:25:00.000-05:00</m:dateAndTime>
  </m:reservation>
 </env:Header>
 <env:Body>
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
            xmlns:x="http://travelcompany.example.org/vocab#"
     env:encodingStyle="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <x:ReservationRequest
 rdf:about="http://travelcompany.example.org/reservations?code=FT35ZBQ">
      <x:passenger>Åke Jógvan Øyvind</x:passenger>
      <x:outbound>
       <x:TravelRequest>
        <x:to>LAX</x:to>
        <x:from>LGA</x:from>
        <x:date>2001-12-14</x:date>
       </x:TravelRequest>
      </x:outbound>
      <x:return>
       <x:TravelRequest>
        <x:to>JFK</x:to>
        <x:from>LAX</x:from>
        <x:date>2001-12-20</x:date>
       </x:TravelRequest>
      </x:return>
    </x:ReservationRequest>
  </rdf:RDF>
 </env:Body>
</env:Envelope>
```

SOAP message returned as a response to the HTTP GET in Example 8a

Note that the reservation details could well have been returned as an (X)HTML document, but this example wanted to show a case where the reservation application is returning the state of the resource (the reservation) in a data-centric media form (a SOAP message) which can be machine processed, instead of (X)HTML which would be processed by a browser. Indeed, in the most likely anticipated uses of SOAP, the consuming application will not be a browser.

Also, as shown in the example, the use of SOAP in the HTTP response body offers the possibility of expressing some application-specific feature through the use of SOAP headers. By using SOAP, the application is provided with a useful and consistent framework and processing model for expressing such features.

4.1.2 SOAP HTTP POST Usage

Using the HTTP binding with the SOAP Request-Response message exchange pattern is restricted to the HTTP POST method. Note that the use of this message exchange pattern in the SOAP HTTP binding is available to all applications, whether they involve the exchange of general XML data or RPCs (as in the following examples) encapsulated in SOAP messages.

Examples 9 and 10 show an example of a HTTP binding using the SOAP Request-Response message exchange pattern, using the same scenario as that for Example 4 and Example 5a, respectively, namely conveying an RPC and its return in the body of a SOAP message. The examples and discussion in this section only concentrate on the HTTP headers and their role.

Example 9

```
POST /Reservations HTTP/1.1
Host: travelcompany.example.org
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
 <env:Header>
   <t:transaction
           xmlns:t="http://thirdparty.example.org/transaction"
           env:encodingStyle="http://example.com/encoding"
           env:mustUnderstand="true" >5</t:transaction>
 </env:Header>
 <env:Body>
  <m:chargeReservation
     env:encodingStyle="http://www.w3.org/2003/05/soap-encoding"
          xmlns:m="http://travelcompany.example.org/">
   <m:reservation xmlns:m="http://travelcompany.example.org/reservation">
    <m:code>FT35ZBQ</m:code>
   </m:reservation>
    <o:creditCard xmlns:o="http://mycompany.example.com/financial">
     <n:name xmlns:n="http://mycompany.example.com/employees">
           Åke Jógvan Øyvind
     </n:name>
     <o:number>123456789099999</o:number>
     <o:expiration>2005-02</o:expiration>
    </o:creditCard>
   </m:chargeReservation>
  </env:Body>
</env:Envelope>
```

RPC in Example 4 carried in an HTTP POST Request

Example 9 shows an RPC request directed at the travel service application. The SOAP message is sent in the body of a HTTP POST method directed at the URI identifying the "Reservations" resource on the server travelcompany.example.org. When using HTTP, the Request-URI indicates the resource to which the invocation is "posted". Other than requiring that it be a valid URI, SOAP places no *formal* restriction on the form of the request URI (see [RFC 2396] for more information on URIs). However, one of the principles of the Web architecture is that all important resources be identified by URIs. This implies that most well-architected SOAP services will be embodied as a large number of resources, each with its own URI. Indeed, many such resources are likely to be created dynamically during the operation of a service, such as, for instance, the specific travel reservation shown in the example. So, a well-architected travel service application should have different URIs for each reservation, and SOAP requests to retrieve or manipulate those reservations will be directed at their URIs, and not at a single monolithic "Reservations" URI, as shown in Example 9. Example 13 in section 4.1.3 shows the preferred way to address resources such as a particular travel reservation. Therefore, we defer until section 4.1.3 further discussion of Web architecture compatible SOAP/HTTP usage.

When placing SOAP messages in HTTP bodies, the HTTP Content-type header must be chosen as "application/soap+xml". (The optional charset parameter, which can take the value of "utf-8" or "utf-16", is shown in this example, but if it is absent the character set rules for freestanding [XML 1.0] apply to the body of the HTTP request.)

Example 10 shows the RPC return (with details omitted) sent by the travel service application in the

corresponding HTTP response to the request from Example 5a. SOAP, using HTTP transport, follows the semantics of the HTTP status codes for communicating status information in HTTP. For example, the 2xx series of HTTP status codes indicate that the client's request (including the SOAP component) was successfully received, understood, and accepted etc.

Example 10

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
 <env:Header>
       ...
       ...
 </env:Header>
 <env:Body>
       ...
       ...
 </env:Body>
</env:Envelope>
```

RPC return in Example 5a embedded in an HTTP Response indicating a successful completion

If an error occurs processing the request, the HTTP binding specification requires that a HTTP 500 "Internal Server Error" be used with an embedded SOAP message containing a SOAP fault indicating the server-side processing error.

Example 11 is the same SOAP fault message as Example 6a, but this time with the HTTP headers added.

Example 11

```
HTTP/1.1 500 Internal Server Error
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Body>
    <env:Fault>
     <env:Code>
       <env:Value>env:Sender</env:Value>
       <env:Subcode>
        <env:Value>rpc:BadArguments</env:Value>
       </env:Subcode>
     </env:Code>
     <env:Reason>
      <env:Text xml:lang="en-US">Processing error</env:Text>
      <env:Text xml:lang="cs">Chyba zpracování</env:Text>
     </env:Reason>
     <env:Detail>
      <e:myFaultDetails
        xmlns:e="http://travelcompany.example.org/faults" >
        <e:message>Name does not match card number</e:message>
        <e:errorcode>999</e:errorcode>
      </e:myFaultDetails>
     </env:Detail>
   </env:Fault>
 </env:Body>
</env:Envelope>
```

Sample SOAP message in a HTTP Response indicating failure to handle the SOAP Body in Example 4

SOAP Part 2 Table 16 provides detailed behavior for handling the various possible HTTP response codes, i.e., the 2xx (successful), 3xx (redirection), 4xx (client error) and 5xx (server error).

4.1.3 Web Architecture Compatible SOAP Usage

One of the most central concepts of the World Wide Web is that of a URI as a resource identifier. SOAP

services that use the HTTP binding and wish to interoperate with other Web software should use URIs to address all important resources in their service. For example, a very important - indeed predominant - use of the World Wide Web is pure information retrieval, where the representation of an available resource, identified by a URI, is fetched using a HTTP GET request without affecting the resource in any way. (This is called a safe and idempotent method in HTTP terminology.) The key point is that the publisher of a resource makes available its URI, which consumers may "GET".

There are many instances when SOAP messages are designed for uses which are purely for information retrieval, such as when the state of some resource (or object, in programming terms) is requested, as opposed to uses that perform resource manipulation. In such instances, the use of a SOAP body to carry the request for the state, with an element of the body representing the object in question, is seen as counter to the spirit of the Web because the resource is not identified by the Request-URI of the HTTP GET. (In some SOAP/RPC implementations, the HTTP Request-URI is often not the identifier of the resource itself but some intermediate entity which has to evaluate the SOAP message to identify the resource.)

To highlight the changes needed, Example 12a shows the way that is **not** recommended for doing safe information retrieval on the Web. This is an example of an RPC carried in a SOAP message, again using the travel reservation theme, where the request is to retrieve the itinerary for a particular reservation identified by one of the parameters, `reservationCode`, of the RPC. (For purposes of this discussion, it is assumed that the application using this RPC request does not need features which require the use of SOAP headers.)

Example 12a

```
POST /Reservations HTTP/1.1
Host: travelcompany.example.org
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
  <env:Body>
    <m:retrieveItinerary
        env:encodingStyle="http://www.w3.org/2003/05/soap-encoding"
             xmlns:m="http://travelcompany.example.org/">
      <m:reservationCode>FT35ZBQ</m:reservationCode>
    </m:retrieveItinerary>
  </env:Body>
</env:Envelope>
```

This representation is discouraged in cases where the operation is a "safe" retrieval (i.e., it has no side effects)

Note that the resource to be retrieved is not identified by the target URI in the HTTP request but has to be obtained by looking within the SOAP envelope. Thus, it is not possible, as would be the case with other "gettable" URIs on the Web, to make this available via HTTP alone to consumers on the World Wide Web.

SOAP Part 2 section 4.1 offers recommendations on how RPCs that constitute safe and idempotent information retrievals may be defined in a Web-friendly manner. It does so by distinguishing aspects of the method and specific parameters in an RPC definition that serve to identify resources from those that serve other purposes. In Example 12a, the resource to be retrieved is identified by two things: the first is that it is an itinerary (part of the method name), and the second is the reference to a specific instance (a parameter to the method). In such a case, the recommendation is that these resource-identifying parts be made available in the HTTP Request-URI identifying the resource, as for example, as follows:
`http://travelcompany.example.org/reservations/itinerary?reservationCode=FT35ZBQ`.

Furthermore, when an RPC definition is such that all parts of its method description can be described as resource-identifying, the entire target of the RPC may be identified by a URI. In this case, if the supplier of the resource can also assure that a retrieval request is safe, then SOAP Version 1.2 recommends that the choice of the Web method property of GET and the use of the SOAP Response message exchange pattern be used as described in section 4.1.1. This will ensure that the SOAP RPC is performed in a Web architecture compatible manner. Example 12b shows the preferred way for a SOAP node to request the safe retrieval of a resource.

Example 12b

```
GET /Reservations/itinerary?reservationCode=FT35ZBQ  HTTP/1.1
Host: travelcompany.example.org
Accept: application/soap+xml
```

The Web architecture compatible alternative to representing the RPC in Example 12a

It should be noted that SOAP Version 1.2 does not specify any algorithm on how to compute a URI from the definition of an RPC which has been determined to represent pure information retrieval.

Note, however, that if the application requires the use of features that can only have a binding-specific expression within the SOAP infoset, i.e., using SOAP header blocks, then the application must choose HTTP POST method with a SOAP message in the request body.

It also requires the use of the SOAP Request-Response message exchange pattern implemented via a HTTP POST if the RPC description includes data (parameters) which are not resource-identifying. Even in this case, the HTTP POST with a SOAP message can be represented in a Web-friendly manner. As with the use of the GET, [SOAP Part2] recommends for the general case that any part of the SOAP message that serves to identify the resource to which the request is POSTed be identified in the HTTP Request-URI. The same parameters may, of course, be retained in the SOAP `env:Body` element. (The parameters must be retained in the Body in the case of a SOAP-based RPC as these are related to the procedure/method description expected by the receiving application.)

Example 13 is the same as that in Example 9, except that the HTTP Request-URI has been modified to include the reservation `code`, which serves to identify the resource (the reservation in question, which is being confirmed and paid for).

Example 13

```
POST /Reservations?code=FT35ZBQ HTTP/1.1
Host: travelcompany.example.org
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn

<?xml version='1.0'?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
 <env:Header>
   <t:transaction
           xmlns:t="http://thirdparty.example.org/transaction"
           env:encodingStyle="http://example.com/encoding"
           env:mustUnderstand="true" >5</t:transaction>
 </env:Header>
 <env:Body>
  <m:chargeReservation
      env:encodingStyle="http://www.w3.org/2003/05/soap-encoding"
         xmlns:m="http://travelcompany.example.org/">
   <m:reservation xmlns:m="http://travelcompany.example.org/reservation">
    <m:code>FT35ZBQ</m:code>
   </m:reservation>
   <o:creditCard xmlns:o="http://mycompany.example.com/financial">
    <n:name xmlns:n="http://mycompany.example.com/employees">
           Åke Jógvan Øyvind
    </n:name>
    <o:number>123456789099999</o:number>
    <o:expiration>2005-02</o:expiration>
   </o:creditCard>
  </m:chargeReservation>
 </env:Body>
</env:Envelope>
```

RPC from Example 4 carried in an HTTP POST Request in a Web-friendly manner

In Example 13, the resource to be manipulated is identified by two things: the first is that it is a reservation (part of the method name), and the second is the specific instance of a reservation (which is the value of the parameter `code` to the method). The remainder of the parameters in the RPC such as the `creditCard` number are not resource-identifying, but are ancillary data to be processed by the resource. It is the recommendation of [SOAP Part2] that resources that may be accessed by SOAP-based RPCs should, where practical, place any such resource-identifying information as a part of the URI identifying the target of the RPC. It should be noted, however, that [SOAP Part2] does not offer any algorithm to do so. Such algorithms may be developed

in future. Note, however, that all the resource-identifying elements have been retained as in Example 9 in their encoded form in the SOAP `env:Body` element.

In other words, as seen from the above examples, the recommendation in the SOAP specifications is to use URIs in a Web-architecture compatible way - that is, as resource identifiers - whether or not it is GET or POST that is used.

4.2 SOAP Over Email

Application developers can use the Internet email infrastructure to move SOAP messages as either email text or attachments. The examples shown below offer one way to carry SOAP messages, and should not be construed as being the standard way of doing so. The SOAP Version 1.2 specifications do not specify such a binding. However, there is a *non-normative* W3C Note [SOAP Email Binding] describing an email binding for SOAP, its main purpose being to demonstrate the application of the general SOAP Protocol Binding Framework described in [SOAP Part 1].

Example 14 shows the travel reservation request message from Example 1 carried as an email message between a sending and receiving mail user agent. It is implied that the receiver node has SOAP capabilities, to which the body of the email is delivered for processing. (It is assumed that the sending node also has SOAP capabilities so as to be able to process any SOAP faults received in response, or to correlate any SOAP messages received in response to this one.)

Example 14

```
From: a.oyvind@mycompany.example.com
To: reservations@travelcompany.example.org
Subject: Travel to LA
Date: Thu, 29 Nov 2001 13:20:00 EST
Message-Id: <EE492E16A090090276D208424960C0C@mycompany.example.com>
Content-Type: application/soap+xml

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
         env:mustUnderstand="true">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</reference>
   <m:dateAndTime>2001-11-29T13:20:00.000-05:00</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n="http://mycompany.example.com/employees"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
         env:mustUnderstand="true">
   <n:name>Åke Jógvan Øyvind</n:name>
  </n:passenger>
 </env:Header>
 <env:Body>
  <p:itinerary
     xmlns:p="http://travelcompany.example.org/reservation/travel">
   <p:departure>
     <p:departing>New York</p:departing>
     <p:arriving>Los Angeles</p:arriving>
     <p:departureDate>2001-12-14</p:departureDate>
     <p:departureTime>late afternoon</p:departureTime>
     <p:seatPreference>aisle</p:seatPreference>
   </p:departure>
   <p:return>
     <p:departing>Los Angeles</p:departing>
     <p:arriving>New York</p:arriving>
     <p:departureDate>2001-12-20</p:departureDate>
     <p:departureTime>mid morning</p:departureTime>
     <p:seatPreference/>
   </p:return>
  </p:itinerary>
  <q:lodging
     xmlns:q="http://travelcompany.example.org/reservation/hotels">
   <q:preference>none</q:preference>
  </q:lodging>
 </env:Body>
</env:Envelope>
```

SOAP message from Example 1 carried in a SMTP message

The header in Example 14 is in the standard form [RFC 2822] for email messages.

Although an email is a one-way message exchange, and no guarantee of delivery is provided, email infrastructures like the Simple Mail Transport Protocol (SMTP) specification [SMTP] offer a delivery notification mechanism which, in the case of SMTP, are called Delivery Status Notification (DSN) and Message Disposition Notification (MDN). These notifications take the form of email messages sent to the email address specified in the mail header. Applications, as well as email end users, can use these mechanisms to provide the status of an email transmission, but these, if delivered, are notifications at the SMTP level. The application developer must fully understand the capabilities and limitations of these delivery notifications or risk assuming a successful data delivery when none occurred.

SMTP delivery status messages are separate from message processing at the SOAP layer. Resulting SOAP responses to the contained SOAP data will be returned through a new email message which may or may not have a link to the original requesting email at the SMTP level. The use of the [RFC 2822] `In-reply-to:` header can achieve a correlation at the SMTP level, but does not necessarily offer a correlation at the SOAP level.

Example 15 is exactly the same scenario as described for Example 2, which shows the SOAP message (body details omitted for brevity) sent from the travel service application to the travel reservation application seeking clarification on some reservation details, except that it is carried as an email message. In this example, the original email's `Message-Id` is carried in the additional email header `In-reply-to:`, which correlates email messages at the SMTP level, but cannot provide a SOAP-specific correlation. In this example, the application relies on the `reservation` header block to correlate SOAP messages. Again, how such correlation is achieved is application-specific, and is not within the scope of SOAP.

Example 15

```
From: reservations@travelcompany.example.org
To: a.oyvind@mycompany.example.com
Subject: Which NY airport?
Date: Thu, 29 Nov 2001 13:35:11 EST
Message-Id: <200109251753.NAA10655@travelcompany.example.org>
In-reply-to:<EE492E16A090090276D208424960C0C@mycompany.example.com>
Content-Type: application/soap+xml

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
     env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
       env:mustUnderstand="true">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</reference>
   <m:dateAndTime>2001-11-29T13:35:00.000-05:00</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n="http://mycompany.example.com/employees"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
        env:mustUnderstand="true">
   <n:name>Åke Jógvan Øyvind</n:name>
  </n:passenger>
 </env:Header>
 <env:Body>
  <p:itinerary
     xmlns:p="http://travelcompany.example.org/reservation/travel">
   <p:itineraryClarifications>
      ...
      ...
   </p:itineraryClarifications>
  </p:itinerary>
 </env:Body>
</env:Envelope>
```

SOAP message from Example 2 carried in an email message with a header correlating it to a previous message.

5. Advanced Usage Scenarios

5.1 Using SOAP Intermediaries

The travel reservation scenario used throughout the primer offers an opportunity to expose some uses of SOAP intermediaries. Recall that the basic exchange was the exchange of a travel reservation request between a travel reservation application and a travel service application. SOAP does not specify how such a message path is determined and followed. That is outside the scope of the SOAP specification. It does describe, though, how a SOAP node should behave if it receives a SOAP message for which it is not the ultimate receiver. SOAP Version 1.2 describes two types of intermediaries: forwarding intermediaries and active intermediaries.

A forwarding intermediary is a SOAP node which, based on the semantics of a header block in a received SOAP message or based on the message exchange pattern in use, forwards the SOAP message to another SOAP node. For example, processing a "routing" header block describing a message path feature in an incoming SOAP message may dictate that the SOAP message be forwarded to another SOAP node identified by data in that header block. The format of the SOAP header of the outbound SOAP message, i.e., the placement of inserted or reinserted header blocks, is determined by the overall processing at this *forwarding* intermediary based on the semantics of the processed header blocks.

An active intermediary is one that does additional processing on an incoming SOAP message before forwarding the message using criteria that are not described by incoming SOAP header blocks, or by the message exchange pattern in use. Some examples of such active intervention at a SOAP node could be, for instance, encrypting some parts of a SOAP message and providing the information on the cipher key in a header block, or including some additional information in a new header block in the outbound message providing a timestamp or an annotation, for example, for interpretation by appropriately targeted nodes downstream.

One mechanism by which an active intermediary can describe the modifications performed on a message is by inserting header blocks into the outbound SOAP message. These header blocks can inform downstream SOAP nodes acting in roles whose correct operation depends on receiving such notification. In this case, the semantics of such inserted header blocks should also call for either the same or other header blocks to be (re)inserted at subsequent intermediaries as necessary to ensure that the message can be safely processed by nodes yet further downstream. For example, if a message with header blocks removed for encryption passes through a second intermediary (without the original header blocks being decrypted and reconstructed), then indication that the encryption has occurred must be retained in the second relayed message.

In the following example, a SOAP node is introduced in the message path between the travel reservation and travel service applications, which intercepts the message shown in Example 1. An example of such a SOAP node is one which logs all travel requests for off-line review by a corporate travel office. Note that the header blocks `reservation` and `passenger` in that example are intended for the node(s) that assume the role "next", which means that it is targeted at the next SOAP node in the message path that receives the message. The header blocks are mandatory (the `mustUnderstand` attribute is set to "true"), which means that the node must have knowledge (through an external specification of the header blocks' semantics) of what to do. A logging specification for such header blocks might simply require that various details of the message be recorded at every node that receives such a message, and that the message be relayed along the message path unchanged. (Note that the specifications of the header blocks must require that the same header blocks be reinserted in the outbound message, because otherwise, the SOAP processing model would require that they be removed.) In this case, the SOAP node acts as a forwarding intermediary.

A more complex scenario is one where the received SOAP message is amended in some way not anticipated by the initial sender. In the following example, it is assumed that a corporate travel application at the SOAP intermediary attaches a header block to the SOAP message from Example 1 before relaying it along its message path towards the travel service application - the ultimate recipient. The header block contains the constraints imposed by a travel policy for this requested trip. The specification of such a header block might require that the ultimate recipient (and only the ultimate recipient, as implied by the absence of the `role` attribute) make use of the information conveyed by it when processing the body of the message.

Example 16 shows an active intermediary inserting an additional header block, `travelPolicy`, intended for the ultimate recipient which includes information that qualifies the application-level processing of this travel request.

Example 16

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
     env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
        env:mustUnderstand="true">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</reference>
   <m:dateAndTime>2001-11-29T13:20:00.000-05:00</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n="http://mycompany.example.com/employees"
     env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
       env:mustUnderstand="true">
   <n:name>Åke Jógvan Øyvind</n:name>
  </n:passenger>
  <z:travelPolicy
    xmlns:z="http://mycompany.example.com/policies"
      env:mustUnderstand="true">
   <z:class>economy</z:class>
   <z:fareBasis>non-refundable<z:fareBasis>
   <z:exceptions>none</z:exceptions>
  </z:travelPolicy>
 </env:Header>
 <env:Body>
  <p:itinerary
    xmlns:p="http://travelcompany.example.org/reservation/travel">
   <p:departure>
     <p:departing>New York</p:departing>
     <p:arriving>Los Angeles</p:arriving>
     <p:departureDate>2001-12-14</p:departureDate>
     <p:departureTime>late afternoon</p:departureTime>
     <p:seatPreference>aisle</p:seatPreference>
   </p:departure>
   <p:return>
     <p:departing>Los Angeles</p:departing>
     <p:arriving>New York</p:arriving>
     <p:departureDate>2001-12-20</p:departureDate>
     <p:departureTime>mid morning</p:departureTime>
     <p:seatPreference/>
   </p:return>
  </p:itinerary>
  <q:lodging
    xmlns:q="http://travelcompany.example.org/reservation/hotels">
   <q:preference>none</q:preference>
  </q:lodging>
 </env:Body>
</env:Envelope>
```

SOAP message from Example 1 for a travel reservation after an active intermediary has inserted a mandatory header intended for the ultimate recipient of the message

5.2 Using Other Encoding Schemes

Even though SOAP Version 1.2 defines a particular encoding scheme (see SOAP Part 2 section 3), its use is optional and the specification makes clear that other encoding schemes may be used for application-specific data within a SOAP message. For this purpose it provides the attribute `env:encodingStyle`, of type `xs:anyURI`, to qualify header blocks, any child elements of the SOAP `env:Body`, and any child elements of the `env:Detail` element and their descendants. It signals a serialization scheme for the nested contents, or at least the one in place until another element is encountered which indicates another encoding style for its nested contents. The choice of the value for the `env:encodingStyle` attribute is an application-specific decision and the ability to interoperate is assumed to have been settled "out-of-band". If this attribute is not present, then no claims are being made about the encoding being used.

The use of an alternative encoding scheme is illustrated in Example 17. Continuing with the travel reservation theme, this example shows a SOAP message which is sent to the passenger from the travel service after the reservation is confirmed, showing the travel details. (The same message was used in Example 8b in another context.)

Example 17

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
     env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
        env:mustUnderstand="true">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
   <m:dateAndTime>2001-11-30T16:25:00.000-05:00</m:dateAndTime>
  </m:reservation>
 </env:Header>
 <env:Body>
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
            xmlns:x="http://travelcompany.example.org/vocab#"
    env:encodingStyle="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <x:ReservationRequest
 rdf:about="http://travelcompany.example.org/reservations?code=FT35ZBQ">
      <x:passenger>Åke Jógvan Øyvind</x:passenger>
      <x:outbound>
       <x:TravelRequest>
        <x:to>LAX</x:to>
        <x:from>LGA</x:from>
        <x:date>2001-12-14</x:date>
       </x:TravelRequest>
      </x:outbound>
      <x:return>
       <x:TravelRequest>
        <x:to>JFK</x:to>
        <x:from>LAX</x:from>
        <x:date>2001-12-20</x:date>
       </x:TravelRequest>
      </x:return>
    </x:ReservationRequest>
  </rdf:RDF>
 </env:Body>
</env:Envelope>
```

SOAP message showing the use of an alternative encoding for the `Body` element

In Example 17, the body of the SOAP message contains a description of the itinerary using the encoding of a graph of resources and their properties using the syntax of the Resource Description Framework (RDF) [RDF]. (Very briefly, as RDF syntax or usage is not the subject of this primer, an RDF graph relates resources - such as the travel reservation resource available at `http://travelcompany.example.org/reservations?code=FT35ZBQ` - to other resources (or values) via properties, such as the `passenger`, the `outbound` and `return` dates of travel. The RDF encoding for the itinerary might have been chosen, for example, to allow the passenger's travel application to store it in an RDF-capable calendar application, which could then be queried in complex ways.)

6. Changes Between SOAP 1.1 and SOAP 1.2

SOAP Version 1.2 has a number of changes in syntax and provides additional (or clarified) semantics from those described in [SOAP 1.1]. The following is a list of features where the two specifications differ. The purpose of this list is to provide the reader with a quick and easily accessible summary of the differences between the two specifications. The features have been put in categories purely for ease of reference, and in some cases, an item might equally well have been placed in another category.

Document structure

- The SOAP 1.2 specifications have been provided in two parts. [SOAP Part1] provides an abstract Infoset-based definition of the SOAP message structure, a processing model and an underlying protocol binding framework, while [SOAP Part2] provides serialization rules for conveying that infoset as well as a particular HTTP binding.
- SOAP 1.2 will not spell out the acronym.
- SOAP 1.2 has been rewritten in terms of XML infosets, and not as serializations of the form <?xml....?> required by SOAP 1.1.

Additional or changed syntax

- SOAP 1.2 does not permit any element after the body. The SOAP 1.1 schema definition allowed for

such a possibility, but the textual description is silent about it.
- SOAP 1.2 does not allow the `env:encodingStyle` attribute to appear on the SOAP `env:Envelope`, whereas SOAP 1.1 allows it to appear on any element. SOAP 1.2 specifies specific elements where this attribute may be used.
- SOAP 1.2 defines the new `env:NotUnderstood` header element for conveying information on a mandatory header block which could not be processed, as indicated by the presence of an `env:MustUnderstand` fault code. SOAP 1.1 provided the fault code, but no details on its use.
- In the SOAP 1.2 infoset-based description, the `env:mustUnderstand` attribute in header elements takes the (logical) value "true" or "false", whereas in SOAP 1.1 they are the literal value "1" or "0" respectively.
- SOAP 1.2 provides a new fault code `DataEncodingUnknown`.
- The various namespaces defined by the two protocols are of course different.
- SOAP 1.2 replaces the attribute `env:actor` with `env:role` but with essentially the same semantics.
- SOAP 1.2 defines a new attribute, `env:relay`, for header blocks to indicate if unprocessed header blocks should be forwarded.
- SOAP 1.2 defines two new roles, "none" and "ultimateReceiver", together with a more detailed processing model on how these behave.
- SOAP 1.2 has removed the "dot" notation for fault codes, which are now simply an XML Qualified Name, where the namespace prefix is the SOAP envelope namespace.
- SOAP 1.2 replaces "client" and "server" fault codes with "Sender" and "Receiver".
- SOAP 1.2 uses the element names `env:Code` and `env:Reason`, respectively, for what used to be called `faultcode` and `faultstring` in SOAP 1.1. SOAP 1.2 also allows multiple `env:Text` child elements of `env:Reason` qualified by `xml:lang` to allow multiple language versions of the fault reason.
- SOAP 1.2 provides a hierarchical structure for the mandatory SOAP `env:Code` sub-element in the `env:Fault` element, and introduces two new optional subelements, `env:Node` and `env:Role`.
- SOAP 1.2 removes the distinction that was present in SOAP 1.1 between header and body faults as indicated by the presence of the `env:Details` element in `env:Fault`. In SOAP 1.2, the presence of the `env:Details` element has no significance as to which part of the fault SOAP message was processed.
- SOAP 1.2 uses XML Base [XML Base] for determining a base URI for relative URI references whereas SOAP 1.1 is silent about the matter.

SOAP HTTP binding

- In the SOAP 1.2 HTTP binding, the `SOAPAction` HTTP header defined in SOAP 1.1 has been removed, and a new HTTP status code 427 has been sought from IANA for indicating (at the discretion of the HTTP origin server) that its presence is required by the server application. The contents of the former `SOAPAction` HTTP header are now expressed as a value of an (optional) "action" parameter of the "application/soap+xml" media type that is signaled in the HTTP binding.
- In the SOAP 1.2 HTTP binding, the Content-type header should be "application/soap+xml" instead of "text/xml" as in SOAP 1.1. The IETF registration for this new media type is pending [SOAP MediaType].
- SOAP 1.2 provides a finer grained description of use of the various 2xx, 3xx, 4xx HTTP status codes.
- Support of the HTTP extensions framework has been removed from SOAP 1.2.
- SOAP 1.2 provides an additional message exchange pattern which may be used as a part of the HTTP binding that allows the use of HTTP GET for safe and idempotent information retrievals.

RPC

- SOAP 1.2 provides a `rpc:result` element accessor for RPCs.
- SOAP 1.2 provides several additional fault codes in the RPC namespace.
- SOAP 1.2 offers guidance on a Web-friendly approach to defining RPCs where the procedure's purpose is purely "safe" informational retrieval.

SOAP encodings

- An abstract data model based on a directed edge labeled graph has been formulated for SOAP 1.2. The SOAP 1.2 encodings are dependent on this data model. The SOAP RPC conventions are dependent on this data model, but have no dependencies on the SOAP encoding. Support of the SOAP 1.2 encodings and SOAP 1.2 RPC conventions are optional.
- The syntax for the serialization of an array has been changed in SOAP 1.2 from that in SOAP 1.1.
- The support provided in SOAP 1.1 for partially transmitted and sparse arrays is not available in SOAP 1.2.
- SOAP 1.2 allows the inline (embedded) serialization of multiref values.

- The `href` attribute in SOAP 1.1 (of type `xs:anyURI`) is called `enc:ref` in SOAP 1.2 and is of type `IDREF`.
- In SOAP 1.2, omitted accessors of compound types are made equal to NILs.
- SOAP 1.2 provides several fault sub-codes for indicating encoding errors.
- Types on nodes are made optional in SOAP 1.2.
- SOAP 1.2 has removed generic compound values from the SOAP Data Model.
- SOAP 1.2 has added an optional attribute `enc:nodeType` to elements encoded using SOAP encoding that identifies its structure (i.e., a simple value, a struct or an array).

SOAP Part 1 Appendix A provides version management rules for a SOAP node that can support the version transition from [SOAP 1.1] to SOAP Version 1.2. In particular, in defines an `env:Upgrade` header block which can be used by a SOAP 1.2 node on receipt of a [SOAP 1.1] message to send a SOAP fault message to the originator to signal which version of SOAP it supports.

7. References

[SOAP Part1] W3C Recommendation "SOAP 1.2 Part 1: Messaging Framework", Martin Gudgin, Marc Hadley, Jean-Jacques Moreau, Henrik Frystyk Nielsen, 24 June 2003 (See http://www.w3.org/TR/2003/REC-soap12-part1-20030624/.)

[SOAP Part2] W3C Recommendation "SOAP 1.2 Part 2: Adjuncts", Martin Gudgin, Marc Hadley, Jean-Jacques Moreau, Henrik Frystyk Nielsen, 24 June 2003 (See http://www.w3.org/TR/2003/REC-soap12-part2-20030624.)

[RFC 2396] IETF "RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax", T. Berners-Lee, R. Fielding, L. Masinter, August 1998. (See http://www.ietf.org/rfc/rfc2396.txt.)

[HTTP 1.1] IETF "RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1", R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, T. Berners-Lee, January 1997. (See http://www.ietf.org/rfc/rfc2616.txt.)

[XML 1.0] W3C Recommendation "Extensible Markup Language (XML) 1.0 (Second Edition)", Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, 6 October 2000. (See http://www.w3.org/TR/2000/REC-xml-20001006.

[Namespaces in XML] W3C Recommendation "Namespaces in XML", Tim Bray, Dave Hollander, Andrew Layman, 14 January 1999. (See http://www.w3.org/TR/1999/REC-xml-names-19990114/.)

[XML Schema Part1] W3C Recommendation "XML Schema Part 1: Structures", Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn, 2 May 2001. (See http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/)

[XML Schema Part2] W3C Recommendation "XML Schema Part 2: Datatypes", Paul V. Biron, Ashok Malhotra, 2 May 2001. (See http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/)

[SMTP] SMTP is defined in a series of RFCs:

- RFC 2822 Internet Message Format. (See http://www.ietf.org/rfc/rfc2822.txt.)
- RFC 2045 Multipurpose Internet Mail Extensions (MIME) Part One:Format of Internet Message Bodies. (See http://www.ietf.org/rfc/rfc2045.txt.)
- RFC 2046 Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. (See http://www.ietf.org/rfc/rfc2046.txt.)
- RFC 1894 An Extensible Message Format for Delivery Status Notifications. (See http://www.ietf.org/rfc/rfc1894.txt.)
- RFC 2852 Deliver By SMTP Service Extension. (See http://www.ietf.org/rfc/rfc2852.txt.)
- RFC 2298 An Extensible Message Format for Message Disposition Notifications. (See http://www.ietf.org/rfc/rfc2298.txt.)

[RDF] W3C Recommendation "Resource Description Framework (RDF) Model and Syntax Specification", O. Lassila and R. Swick, Editors. World Wide Web Consortium. 22 February 1999. (See http://www.w3.org/TR/REC-rdf-syntax.)

[SOAP 1.1] W3C Note "Simple Object Access Protocol (SOAP) 1.1", Don Box *et al*., 8 May, 2000 (See

http://www.w3.org/TR/SOAP/)

[XML Infoset] W3C Recommendation "XML Information Set", John Cowan, Richard Tobin, 24 October 2001. (See http://www.w3.org/TR/xml-infoset/)

[SOAP MediaType] IETF Internet Draft "The 'application/soap+xml' media type", M. Baker, M. Nottingham, "draft-baker-soap-media-reg-03.txt", May 29, 2003. (See http://www.ietf.org/internet-drafts/draft-baker-soap-media-reg-03.txt, note that this Internet Draft expires in November 2003)

[SOAP Email Binding] W3C Note "SOAP Version 1.2 Email Binding", Highland Mary Mountain et al, draft June 13, 2002. (See http://www.w3.org/TR/2002/NOTE-soap12-email-20020626.)

[XML Base] W3C Recommendation "XML Base", Jonathan Marsh, 27 June 2001. (See http://www.w3.org/TR/2001/REC-xmlbase-20010627/)

[RFC 2119] IETF "RFC 2119: Key words for use in RFCs to Indicate Requirement Levels", S. Bradner, March 1997. (See http://www.ietf.org/rfc/rfc2119.txt.)

A. Acknowledgements

Highland Mary Mountain (Intel) provided the initial material for the section on the SMTP binding. Paul Denning provided material for a usage scenario, which has since been moved to the SOAP Version 1.2 Usage Scenarios Working Draft. Stuart Williams, Oisin Hurley, Chris Ferris, Lynne Thompson, John Ibbotson, Marc Hadley, Yin-Leng Husband and Jean-Jacques Moreau provided detailed comments on earlier versions of this document, as did many others during the Last Call Working Draft review. Jacek Kopecky provided a list of RPC and SOAP encoding changes.

This document is the work of the W3C XML Protocol Working Group.

Participants in the Working Group are (at the time of writing, and by alphabetical order): Carine Bournez (W3C), Michael Champion (Software AG), Glen Daniels (Macromedia, formerly of Allaire), David Fallside (IBM, Chair), Dietmar Gaertner (Software AG), Tony Graham (Sun Microsystems), Martin Gudgin (Microsoft Corporation, formerly of DevelopMentor), Marc Hadley (Sun Microsystems), Gerd Hoelzing (SAP AG), Oisin Hurley (IONA Technologies), John Ibbotson (IBM), Ryuji Inoue (Matsushita Electric), Kazunori Iwasa (Fujitsu Limited), Mario Jeckle (DaimlerChrysler R. & Tech), Mark Jones (AT&T), Anish Karmarkar (Oracle), Jacek Kopecky (Systinet/Idoox), Yves Lafon (W3C), Michah Lerner (AT&T), Noah Mendelsohn (IBM, formerly of Lotus Development), Jeff Mischkinsky (Oracle), Nilo Mitra (Ericsson), Jean-Jacques Moreau (Canon), Masahiko Narita (Fujitsu Limited), Eric Newcomer (IONA Technologies), Mark Nottingham (BEA Systems, formerly of Akamai Technologies), David Orchard (BEA Systems, formerly of Jamcracker), Andreas Riegg (DaimlerChrysler R. & Tech), Hervé Ruellan (Canon), Jeff Schlimmer (Microsoft Corporation), Miroslav Simek (Systinet/Idoox), Pete Wenzel (SeeBeyond), Volker Wiechers (SAP AG).

Previous participants were: Yasser alSafadi (Philips Research), Bill Anderson (Xerox), Vidur Apparao (Netscape), Camilo Arbelaez (WebMethods), Mark Baker (Idokorro Mobile (Planetfred), formerly of Sun Microsystems), Philippe Bedu (EDF (Electricité de France)), Olivier Boudeville (EDF (Electricité de France)), Don Box (Microsoft Corporation, formerly of DevelopMentor), Tom Breuel (Xerox), Dick Brooks (Group 8760), Winston Bumpus (Novell), David Burdett (Commerce One), Charles Campbell (Informix Software), Alex Ceponkus (Bowstreet), David Chappell (Sonic Software), Miles Chaston (Epicentric), David Clay (Oracle), David Cleary (Progress Software), Conleth O'Connell (Vignette), Ugo Corda (Xerox), Paul Cotton (Microsoft Corporation), Fransisco Cubera (IBM), Jim d'Augustine (eXcelon), Ron Daniel (Interwoven), Dug Davis (IBM), Ray Denenberg (Library of Congress), Paul Denning (MITRE), Frank DeRose (Tibco), Mike Dierken (DataChannel), Andrew Eisenberg (Progress Software), Brian Eisenberg (DataChannel), Colleen Evans (Sonic Software), John Evdemon (XMLSolutions), David Ezell (Hewlett-Packard), Eric Fedok (Active Data Exchange), Chris Ferris (Sun Microsystems), Daniela Florescu (Propel), Dan Frantz (BEA Systems), Michael Freeman (Engenia Software), Scott Golubock (Epicentric), Rich Greenfield (Library of Congress), Hugo Haas (W3C), Mark Hale (Interwoven), Randy Hall (Intel), Bjoern Heckel (Epicentric), Erin Hoffman (Tradia), Steve Hole (MessagingDirect Ltd.), Mary Holstege (Calico Commerce), Jim Hughes (Fujitsu Software Corporation), Yin-Leng Husband (Hewlett-Packard, formerly of Compaq), Scott Isaacson (Novell), Murali Janakiraman (Rogue Wave), Eric Jenkins (Engenia Software), Jay Kasi (Commerce One), Jeffrey Kay (Engenia Software), Richard Koo (Vitria Technology Inc.), Alan Kropp (Epicentric), Julian Kumar (Epicentric), Peter Lecuyer (Progress Software), Tony Lee (Vitria Technology Inc.), Amy Lewis (TIBCO), Bob Lojek (Intalio), Henry Lowe

(OMG), Brad Lund (Intel), Matthew MacKenzie (XMLGlobal Technologies), Murray Maloney (Commerce One), Richard Martin (Active Data Exchange), Highland Mary Mountain (Intel), Alex Milowski (Lexica), Kevin Mitchell (XMLSolutions), Ed Mooney (Sun Microsystems), Dean Moses (Epicentric), Don Mullen (TIBCO), Rekha Nagarajan (Calico Commerce), Raj Nair (Cisco), Mark Needleman (Data Research Associates), Art Nevarez (Novell), Henrik Nielsen (Microsoft Corporation), Kevin Perkins (Compaq), Jags Ramnaryan (BEA Systems), Vilhelm Rosenqvist (NCR), Marwan Sabbouh (MITRE), Waqar Sadiq (Vitria Technology Inc.), Rich Salz (Zolera), Krishna Sankar (Cisco), George Scott (Tradia), Shane Sesta (Active Data Exchange), Lew Shannon (NCR), John-Paul Sicotte (MessagingDirect Ltd.), Simeon Simeonov (Allaire), Simeon Simeonov (Macromedia), Aaron Skonnard (DevelopMentor), Nick Smilonich (Unisys), Soumitro Tagore (Informix Software), James Tauber (Bowstreet), Lynne Thompson (Unisys), Patrick Thompson (Rogue Wave), Jim Trezzo (Oracle), Asir Vedamuthu (WebMethods), Randy Waldrop (WebMethods), Fred Waskiewicz (OMG), David Webber (XMLGlobal Technologies), Ray Whitmer (Netscape), Stuart Williams (Hewlett-Packard), Yan Xu (DataChannel), Amr Yassin (Philips Research), Susan Yee (Active Data Exchange), Jin Yu (Martsoft).

We also wish to thank all the people who have contributed to discussions on xml-dist-app@w3.org.