The Florida State University
College of Arts and Sciences

WSDL Importer

By

Kiran Kaja

Major Professor:

Dr. Robert van Engelen

A project submitted to the department of Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science

The members of the committee approve the Masters project of Kiran Kaja defended on July 26, 2004.

_____

Dr. Robert van Engelen
Supervising Professor

_____

Dr. Xin Yuan
Committee Member

_____

Dr. David Whalley
Committee Member

Acknowledgements:

Thanks to my major professor Dr. Robert van Engelen, for giving me the opportunity to work on this remarkable project. I also want to thank Dr. Xin Yuan and Dr. David Whalley for agreeing to be on my Master's Project defense committee.

Table of Contents

Abstract

Web Services Description Language (WSDL) is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. WSDL is used to describe the methods offered by SOAP Web services. In this project we designed and implemented an automated header file generation tool for C/C++ languages, which takes any well-formed WSDL file and generated the equivalent header files in accordance with GSOAP rules. These header files are used by GSOAP Compiler to generate client and stubs to interface with the server. Thus we completely automate the GSOAP client generation from the WSDL published by the service.

1. Introduction:

Traditionally a program and service were developed to perform a specific task. The set of programs, which interact with each other, was limited and defined at the time of development. But as information technology became widely acceptable this changed. No longer is the set of programs that interact with each other fixed; it keeps on changing from time to time. To enable this, interoperability standards were designed, which allow any program to interact with any other program. A major step in this direction was design of a web service that is a set of functions or procedures, which can be accessed by sending an XML message in SOAP format over HTTP protocol. The discussion about XML and SOAP is given below.

1.1 XML:

XML (eXtensible Markup Language) is defined by [5] as:

> XML is a set of rules for defining semantic tags that break a document into parts and identify the different parts of the document. It is a meta markup language that designs a syntax used to define other domain specific semantic structured markup languages.

XML allows the user to define new tags and the document structure. XML uses schema to describe data. XML with a schema is self-descriptive, cross-platform, software and hardware independent tool for transmitting information. Using XML, data can be exchanged between incompatible systems.

1.2 XML Schema:

An XML schema describes the structure of an XML document, by defining the legal building blocks of an XML document. XML schema defines the data types, default values and fixed values of elements and attributes that can appear in the document.

XML schemas are extensible because they are written in XML. Any XML schema can be reused in other schemas, can derive data types from basic data types and can create multiple schemas from the same document. Although XML schemas are well formed XML, they need additional conditions to be a valid schema. XML schemas secure data

communication, as the sender can describe the data in a way that the receiver will understand.

1.3 XML Namespaces

XML Namespace is defined by [6] as.

> An XML namespace is a collection of names, identified by a URI reference, which are used in XML documents as element types and attribute names. XML namespaces differ from the "namespaces" conventionally used in computing disciplines in that the XML version has internal structure and is not, mathematically speaking, a set.

XML allows users to define their own tags that often lead to name conflicts when two different documents use the same names describing two different types of elements. A prefix added to the elements would resolve the conflict and provide a universally unique XML name. Prefixes are defined using “xmlns” attribute to give the element prefix a qualified name associated with a namespace.

The attribute xmlns:namespace-prefix="namespace" defines a <namespace-prefix> prefix associated with namespace where namespace is a Uniform Resource Identifier (URI). When a namespace is defined in the start tag of an element, all child elements with the same prefix are associated with the same namespace. Any XML name consists of a namespace prefix and a local name to ensure uniqueness.

1.4 DOM

DOM (Document Object Model) is defined by [7] as:

> DOM is an application programming interface (API) for HTML (Hyper Text Markup Language) and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated.

DOM enables programmers to import, modify, and delete any part of a document. DOM also enables the programmer to access any part of the document. In DOM, the document has a logical tree structure representing the hierarchy of the document. Any part of the document can be accessed by traversing the tree structure which allows the modifications to the document structure to be very easy.

1.5 Web Service

A web service is defined by [9] as:

> XML Web services are the fundamental building block in the move to distributed computing on the Internet. Open standards and the focus on communication and collaboration among people and applications have created an environment where XML Web services are becoming the platform for application integration. Applications are constructed using multiple XML Web services from various sources that work together regardless of where they reside or how they were implemented. XML Web services are successful for a variety of reasons, notably: They are based on open standards with multi-vendor support making them interoperable, and the technology used to implement them is ubiquitous.

Web services are based on SOAP, WSDL andUDDI standards.

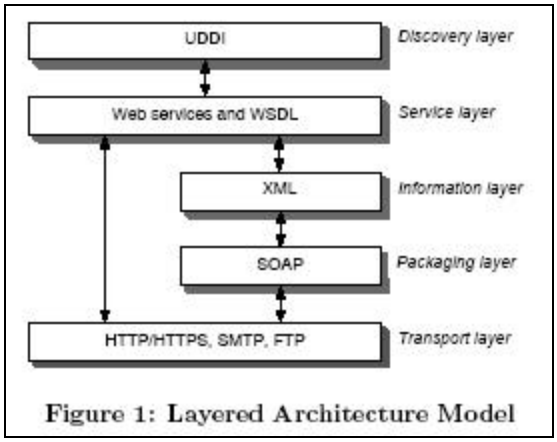

Figure 1: Layered Architecture Model

The *transport layer* is at the bottom of the model. The firewall-friendly HTTP (Hypertext Transfer Protocol) and secure HTTPS are used to invoke Web services with HTTP POST request-response message exchanges.

The *packaging layer* uses the XML-based SOAP protocol. A SOAP message consists of an Envelope root element, an optional SOAP Header to encode meta-information (such as authentication data, signatures, routing points, and transaction tokens), encoding rules defining how messages should be processed, and an RPC representation in the SOAP Body that defines how to represent remote procedure calls and responses, such as the RPC method name and its parameters. The SOAP encoding style uses both scalar types (strings, integers, floats, and so on) and compound types (structures and arrays) that can be used to carry application data. The values of these types appear as XML elements

within the method parameters of the SOAP Body. A SOAP Fault element (not shown) is used to carry error information to transfer remote exceptions.

The *information layer* carries the XML-formatted SOAP message. The process of wrapping application data in XML is called XML serialization. The mechanisms of XML serialization consists of XML encoding to prepare outbound messages for transmission and XML decoding upon receiving inbound messages. To establish a SOAP RPC request-response message exchange, a client invokes a local proxy object. The proxy's RPC stub routine marshals the function parameters in XML, wraps it in a SOAP envelope, and transmits it over the network to be handled remotely, where the reverse process takes place to unwrap the parameters and return a response.

The *services layer* provides meta-data on the interface to Web services as defined by WSDL. WSDL describes a SOAP/XML Web service and promotes reusability by defining the service functionality and access mechanisms, similar to CORBA's IDL for IIOP.

The *discovery layer* offers a way to publish information about web services, as well as provide a mechanism to discover what web services are available through the Universal Description, Discovery, and Integration (UDDI) specification. UDDI is a yellow pages service provider for service registration and lookup.

1.6 XML-RPC:

XML-RPC is defined by [8] as:

> XML-RPC is a Remote Procedure Calling protocol that works over the Internet. An XML-RPC message is an HTTP-POST request. The body of the request is in XML. A procedure executes on the server and the value it returns is also formatted in XML. Procedure parameters can be scalars, numbers, strings, dates, etc.; and can also be complex record and list structures.

XML-RPC is one of the ways in which a client and server can communicate over HTTP. This defines rules for header, payload and the response of a message.

1.7 SOAP:

SOAP (Simple Object Access Protocol) is defined by [1] as:

> 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.

Basically, SOAP is an XML based protocol, which is designed for distributed applications to communicate over HTTP and through firewalls.

1.8 GSOAP:

GSOAP is a C/C++ implementation of SOAP protocol that facilitates creation of SOAP web services and clients. It is a compiler that generates the serialization and de-serialization routines that enable the user to create web services in “pure” C or mixed C/C++. [2].

1.9 WSDL:

WSDL is described by [3] as:

> WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services).

1.10 GSOAP and WSDL:

Each web service is associated with a WSDL (Web Service Descriptor Language) document, which consists of all the functions provided by the web service and their input

and output parameters. GSOAP enables a user to create web services and clients in C and C++. While creating a web service a user has the list of all function he is going to implement but when creating a client the user has no idea of the functions provided by the web service. The only way he can find out about the functions is by analyzing the WSDL file, which is a very tedious task.

1.11 WSDL File structure:

A WSDL file is mainly organized into 5 sections: types (schemas), messages, port type, binding and service. A WSDL file is contained in a definitions tag, which is the root. All nodes are defined in the WSDL namespace.

1.11.1 Types:

User defined data types are defined in a node, called *types*, consisting of multiple schemas. Each schema may further have two different types of nodes, known as SimpleTypes and ComplexTypes. Simple types are used to extend or restrict the values of existing built-in primitive XSD schema types or to define enumerations and enumeration masks. Enumeration values are given by value attributes of enumeration tags. ComplexTypes nodes are used to define aggregate data types that can be loosely compared to arrays, classes and structures. Each complexType node consists of a sequence of element and attribute tags that define the members of the data structure. So-called SOAP encoded data types form a restricted class of XSD schema types. For example, if a complexTypes node consist of a restriction node, with it's base attribute set to 'SOAP-ENC:Array', then that node represents an array of a type given by the value of attribute node's *arrayType* attribute. Otherwise, if that ComplexType node contains an extension node, then it represents an extended class with a base given by the base attribute of the extension node. The name of the data type is given by the name attribute of the node. All the data types defined by element tags make the members of the new data type.

1.11.2 Messages:

Message Nodes are used to represent an operation's input and output messages. Each Message Node consists of nodes known as *Part nodes*. Each Part node of message has a name and type attribute. A Name attribute assigns a name and a Type attribute specifies the data type of the part. A Part node can be of a basic XSD schema or a SOAP-ENC data type or can be a user-defined data type. If a Part node contains an element attribute, then that Part node actually points to a user defined data type.

1.11.3 Port Type:

Port type nodes contain a set of operation nodes. Each operation node further has an input and output node. The message attribute of input or output nodes refer to a pre-defined message node. All the input messages and collection of all input argument messages of the operation node. The output message for the operation is given by the message attribute of the output node.

1.11.4 Binding:

Binding tag, having the name attribute that is defined in the service tag, contains multiple Operation nodes. Each Operation node contains the information about the function defined by the name attribute. An Operation node further has an input, output, fault, soapaction and soapheader nodes. Input, output and fault nodes specify the encoding styles for parameters exchange. The header information extracted from the soapheader node is used as a part of the soap request.

1.11.5 Service:

Service tag contains the name of the service along with its physical location. Service tag also specifies the name of the binding node that contains the operation information for this service. The service information is used to identify the service location and service name.

```
<?xml version="1.0"?>
<definitions name="StockQuote"
               targetNamespace="http://example.com/stockquote.wsdl"
               xmlns:tns="http://example.com/stockquote.wsdl"
               xmlns:xsd1="http://example.com/stockquote.xsd"
               xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
               xmlns="http://schemas.xmlsoap.org/wsdl/">
   <types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
            xmlns="http://www.w3.org/1999/XMLSchema">
     <element name="TradePriceRequest">
      <complexType>
       <all>
        <element name="tickerSymbol" type="string"/>
       </all>
      </complexType>
     </element>
     <element name="TradePrice">
      <complexType>
       <all>
        <element name="price" type="float"/>
       </all>
      </complexType>
     </element>
    </schema>
   </types>

  <message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
  </message>
  <message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
  </message>

  <portType name="StockQuotePortType">
   <operation name="GetLastTradePrice">
    <input message="tns:GetLastTradePriceInput"/>
     <output message="tns:GetLastTradePriceOutput"/>
   </operation>
  </portType>
<binding name="StockQuoteSoapBinding"
type="tns:StockQuotePortType">
  <soap:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetLastTradePrice">
     <soap:operation
      soapAction="http://example.com/GetLastTradePrice"/>
       <input>
         <soap:body use="literal"
          namespace="http://example.com/stockquote.xsd"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
       </input>
      <output>
       <soap:body use="literal"
         namespace="http://example.com/stockquote.xsd"
```

```
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
       </output>
      </operation>
  </binding>
  <service name="StockQuoteService">
   <documentation>My first service</documentation>
    <port name="StockQuotePort" binding="tns:StockQuoteBinding">
     <soap:address location="http://example.com/stockquote"/>
    </port>
  </service>
</definitions>
```

Figure 2 Sample WSDL File

2. Problem Definition:

From discussion of the WSDL file structure, we can see that there is a need for a tool that should enable the user to convert WSDL files to C and C++ header files. The gSOAP compiler generates WSDL documents from header files when a web service is created. But that does not make it possible to generate the header file from the WSDL file. So we believe that a WSDL ***importer*** needs to be designed to complete this cycle. This project presents a solution to this problem.

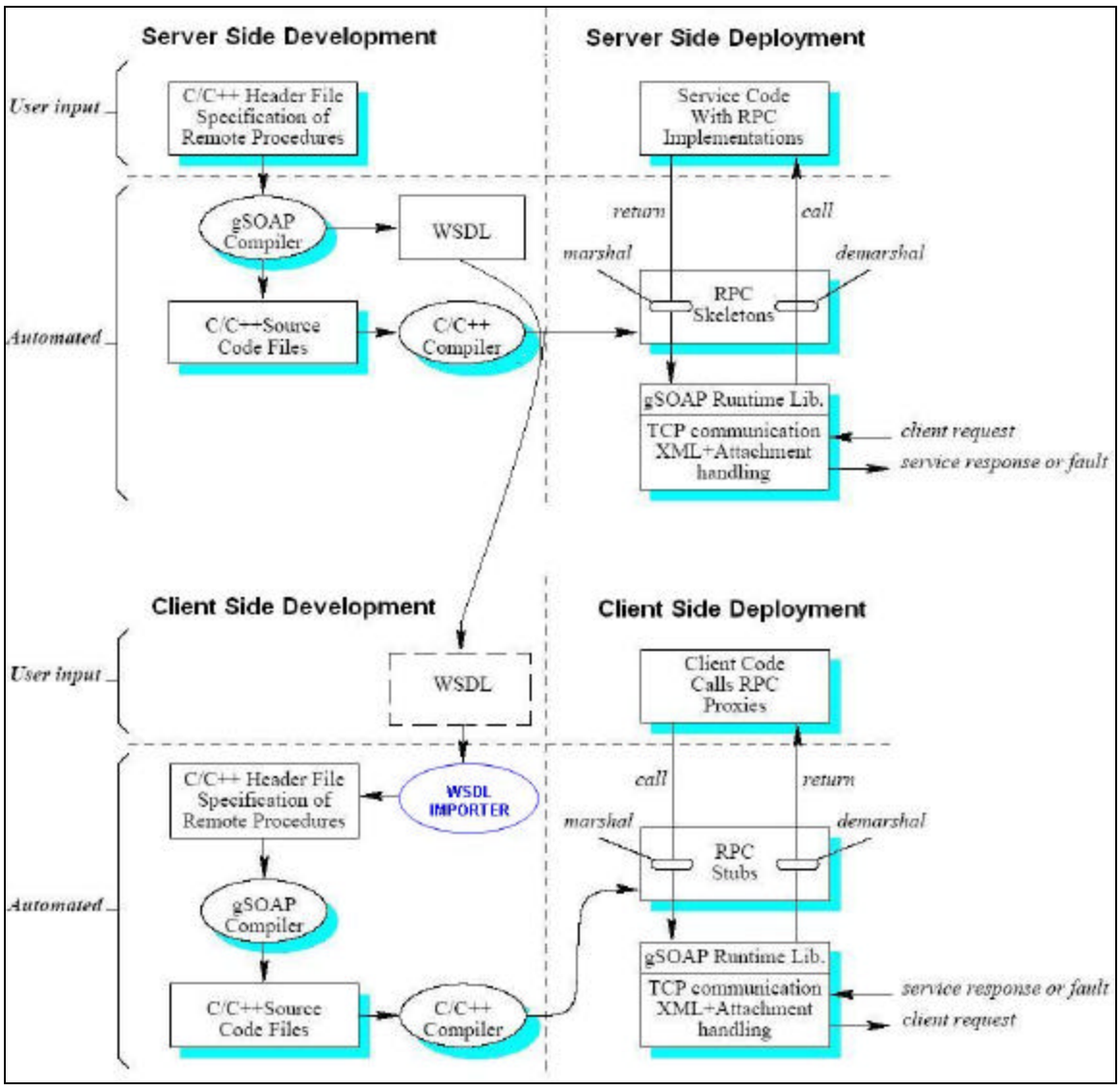

Figure 3 Web Service Deployment Diagram

3. Project Design:

The importer reads a WSDL file and generates a header file for the gSOAP stub and skeleton compiler. The WSDL importer generates a gSOAP header file for any web service described in the WSDL document, which is a valid XML schema defined in WSDL namespace. A WSDL file can be parsed into a DOM tree structure using a DOM parser. The tree structure facilitates easy lookup for required nodes. Generation of data type and operations involves search for specific tags like service, types, operations, bindings etc, identifying valid namespaces, generating the data type and operation definitions in a format to be converted to C and C++ and handling different representations of data types namely RPC and Doc/literal.

WSDL Importer consists of a main control class and 3 helper classes.

1. *NodeSearch* class is built to search for the nodes in a given scope. This class also has the provision to search for nodes with certain specified attributes. Node search takes the root node and the name of element as search arguments to return the array of nodes matching the search conditions.

2. *NSNode* is a wrapper class for the node. It has an additional hash table, which contains the namespace to URI mapping.

3. *NsNodeSearch* class is designed to search for the node along with the namespace information. The search routine adds all the namespaces defined in search path to the hash table corresponding to the node.

Wsdlcpp is the control class. It consists of modules to handle the task of identifying the data types, service primitives and operations.

4. Project Implementation:

The first thing our Importer does is to create a listing of the Definition, Default, WSDL, XSD and SOAP encoding namespaces by looking into the namepspace prefix's defined in the Definition node of WSDL document. The namespace prefix which points to the same URI (Universal Resource Indicator) as the *targetNamespace's* URI, is identified as the Definition namespace for the document if there is match for the URI corresponding to *targetNamespace*, then "defns" is assumed as the Definition namespace for document. The prefix that points to "http://schema.xmlsoap.org/wsdl" is identified as the WSDL namespace. The prefix name space that points to "http://www.w3.org/2001/xmlschema" is identified as the XSD namespace. Finally the prefix that points to "http://schemas.xmlsoap.org/soap/encoding" is identified as the soapencoding namespace. The WSDL document is normally defined under the WSDL namespace.

The XSD namespace is used to identify the default data types. The Soapencoding namespace is used to identify arrays, enumerations and default data types. The Definition namespace is used to declare the functions. All functions are defined in the Definition namespace.

As each node in XML may have different namespace bindings, the namespace hierarchy of the nodes needs to be maintained thorough the search using the NsNodeSearch class. This takes care of namespace overriding and multiple namespaces and also keeps track of XSD, WSDL, DEFAULT and SOAPENC.

Limitation: If some one used more than one namespace of the XSD (for example XSD1 and XSD2) all pointing to the same URI, then the Importer identifies the last defined namespace as the XSD.

4.1 Data Types

All new data types specific to the web service are defined in Types node. Our Importer can generate all possible data types which can be declared in C and C++, that is, basic data types, structures or classes, arrays, enums, enum masks and typedef.

4.1.1 Structure / Classes:

A typical structure / class is shown below:

1. <s:complexType name="Request">
2. <s:sequence>
    a. <s:element minOccurs="0" maxOccurs="unbounded" name="commands" type="s:string" />
    b. <s:element minOccurs="0" maxOccurs="1" name="files" type="s0:ArrayOfRequestFile"/>
    c. <s:element name="variables" nillable = "true" type="s0:ArrayOfRequestVariable" />
3. </s:sequence>
4. </s:complexType>

A structure / class is represented using a ComplexType Node. The *name* attribute gives the name of the data type. The Definition namespace, in which the ComplexType node is defined, gives the namespace prefix for the data type. There is a set of element nodes in ComplexType node, which gives the member variables of the structure.

Each of the element nodes always has a *type* attribute and a *name* attribute. The *name* attribute gives the name and *type* attribute gives the type of member variable. Each member can either be basic or user defined. Data types defined in XSD and the soapencoding namespace are considered to be basic types for which the C/C++ equivalents are pre-defined.

Element Node may also contain *Nillable*, *maxOccurs*, *minOccurs* attributes which are optional. If *Nilliable* is set to be true or if *minOccurs* is 0, then this particular member is optional and may not occur in the XML message, in which case the value of member will have to be set to NULL. To facilitate this, the member will be declared as a pointer. If *maxOccurs* is set to unbound, then the structure can contain any number of member variables of this type. To facilitate this, the member is declared as array by adding a

__size integer variable before the member. To accommodate more than one unbounded member variables, a '_' is appended after the __size variable.

If a member is of basic type, then either XSD or SOAP_ENC (soapencoding) prefix are used instead of its actual name space.

If the member is not basic then it is always declared as a pointer to avoid dependencies. If data type 'A' contains a member of data type B, then B has to be defined before A. This works fine for linear dependences. But in some cases data type B may also contain a member of A, to avoid all these cases all user defined data types members are declared as pointers.

The C++ equivalent of the class above is:

```
class s0__request {
  public:
      int  __size_;
      xsd__string * commands;
      s0__ArrayOfRequestFile * files;
      ArrayOfRequestVariable * variables;
};
```

The C equivalent of the structure above is:

```
struct s0__request {
      int  __size_;
      xsd__string * commands;
      s0__ArrayOfRequestFile * files;
      ArrayOfRequestVariable * variables;
};
```

Explanation: We can observe that the name of the data type is given by the *name* attribute in line 1. The elements nodes in line a to c correspond to the member variables. Line 2.a corresponds to a basic data type, as in this case name space prefix 's' corresponds to the XSD URI. As the *maxOccurs* is set to unbounded, this member actually represents an

array of XSD strings hence a __size_ variable is added before the member. Line 2.b represents a user defined data type variable. The member is of type ArrayOfRequestFile defined in *S0* name space. As the *maxOccurs* is set to 1 this element represents a simple variable. Line 2.c represents a member of user defined array type. The member is identified to be an array type by looking up the all the generated arrays from the wsdl file.

While analyzing the WSDL file, the structures are generated after the arrays. As any of the array members should not have a namespace prefix, arrays defined are identified first and then the structures are generated.

4.1.2 Extended Structures / Classes:

WSDL gives a provision for extending the structures that already exist. Adding an Extension node with a *base* attribute does this. The *base* attribute gives the base class for the current data type. The C and C++ representations differ in case of extended classes. In case of C++, the derived class publicly EXTENDS the base class; but in C the derived structure CONTAINS all the members of the base structure along with the new members. Extended classes are generated after the normal classes as any derived class needs to know the members of the base class. If the base class for a data type is not found in the WSDL file, a warning message is generated.

An example for an extended class is:

1. <complexType name="SendMelodyRequest">
2. <complexContent>
3. <extension base="n:BaseRequest">
4. <sequence>
    a. <element name="melodyID" type="SOAP-ENC:string" />
    b. <element name="from" type="SOAP-ENC:boolean" />
    c. <element name="to" type="xsd:int" />
5. </sequence>
6. </extension>
7. </complexContent>
8. </complexType>

The C++ equivalent is:

```
class n__SendMelodyRequest : public n__BaseRequest {
      SOAP_enc__string  melodyID;
      SOAP_enc__boolean  from;
      xsd__int  to;
};
```

Where the base class is:

```
class n__BaseRequest {
  public:
      xsd__string * applicationID;
};
```

The C equivalent is:

```
struct n__SendMelodyRequest {
      xsd__string * applicationID;
      SOAP_enc__string  melodyID;
      SOAP_enc__boolean  from;
      xsd__int  to;
};
```

Explanation: Because of the Extension node present at line 3, we can observe that the data type is an extended class. The name of the data type is given by the *name* attribute in line1. As the Definition name space for ComplexType node is ‘n’, ‘n__’ is added to the name. The all the member of the data type is given in line 4.a, 4.b and 4.c are a basic data type. As the prefix ‘SOAP-ENC’ prefix corresponds to the soapencoding URI and ‘xsd’ corresponds to the XSD namespace URI.

In C++ the new data type is declared as extension of the BaseRequest. But in case of C all the members (xsd__string * applicationID) of BaseRequest are added to the new data type.

4.1.3 Arrays:

Arrays are identified by the Restriction node with the *base* attribute set to soapencoding array in ComplexType node. The *name* attribute of the ComplexType node gives the name of the array. The type of array is decided by the Attribute node's *arrayType* attribute. The number of open and close bracket pairs in the value of *arrayType* decide the dimensions of the array. Attribute node's *ref* attribute should be set to soapencoding array type and the *arrayType* attribute should be declared in WSDL namespace. The C++ representation of a single-dimensional array is a class with a pointer of the array data type and 2 integer variables. The integer variables are used to store the size and offset of array.

Multi-dimensional arrays are declared as arrays of lower dimensional arrays. It is assumed that the WSDL file contains a one-dimension lower array for every multi-dimensional array. Multi-dimensional arrays are not processed until all single dimensional arrays are generated, as multi-dimensional arrays need the name of the data type equivalent of lower dimension array. In case a lower dimension array is not found, the higher order array is not generated and a warning message is printed.

The main difference between an unbounded attribute of a class / structure and an array is the offset member which is absent in class / structure.

Arrays can be of 3 types:

Array of Basic data type:

> In this case the C++ / C equivalent consists of an XSD or SOAP-ENC pointer variable followed by the integer variables.

Array of User-defined data type:

> In this case the C++ / C equivalent consists of the data type pointer variable qualified by the namespace, size and offset.

Array of Arrays:

In this case the C++ / C equivalent consists of a pointer variable to the array data type. The data type is NOT qualified by any name space prefix. It also consists of size and offset variables.

Array 1 Representation in WSDL

1. <complexType name="ArrayOfdouble" base="SOAP:Array">
2. <complexContent>
    a. <restriction base="SOAP-ENC:Array">
    b. <attribute ref="SOAP-ENC:arrayType" WSDL:arrayType="xsd:double[]"/>
    c. </restriction>
3. </complexContent>
4. </complexType>

Array 2 Representations in WSDL

5. <complexType name="ArrayOfArrayOfdouble" base="SOAP:Array">
6. <complexContent>
    a. <restriction base="SOAP-ENC:Array">
    b. <attribute ref="SOAP-ENC:arrayType" WSDL:arrayType="xsd:double[][]"/>
    c. </restriction>
7. </complexContent>
8. </complexType>

C++ equivalent for Array 1

```
class ArrayOfdouble {
  public:
     xsd__double * __ptr;
     int  __size;
     int  __offset;
};
```

C++ equivalent for Array 2

```
class ArrayOfArrayOfdouble {
  public:
      ArrayOfdouble * __ptr;
      int __size;
      int __offset;
};
```

Explanation: We can identify the ComplexType node at line 1 to be an array due to the restriction node at line 2.a whose *base* attribute is set to soapencoding array. The data type of the array is identified to be XSD double from the value of *arrayType* attribute at line 2.b. The name of the array is given by the *name* attribute in line 1. As there is only one open close bracket pair in the *arrayType* array, this is a single dimensional array.

The ComplexType node at line 5 also represents an array. But as the number of open close brackets pairs in *arrayType* attribute at line 6.a is 2; this is a 2 dimensional XSD double's array. As it is a multidimensional array, the C++ equivalent of this array will be an array of single dimensional XSD double array. We know that "ArrayOfdouble" represents a single dimensional array of XSD double; we can represent our current array as an array of "ArrayOfdouble". So the C++ equivalent will contain a pointer to "ArrayOfdouble" and 2 integer variables to hold the size and offset.

4.1.4 The Typedef Construct:

Many times different data names are used for same data types. This may be needed to give more meaning to the name of data type or to have same data type in different namespace. Using an element node can serve this purpose in WSDL. The *name* attribute gives the name of new data type and the type attribute gives the its equivalent data type.

Equivalents can be declared of either a basic type or a user defined type. Equivalents for basic types are represented using "typedef". But if the equivalent is for user-defined type, a new identical data type is declared containing all the elements of the original data type.

This is done in order to eliminate complications that may arise in extending the extended data types. We can observe that information of user-defined type's will be needed while generating the equivalents, hence Element nodes are processed after user-defined data types.

An example for an equivalent of basic type

1. <s:element name="version" type="s:string" />

An example for an equivalent of user defined data type

2. <s:element name="response" type="s1:Response" />

An example for a C/C++ equivalent:

```
typedef xsd__string  s0__version ;

class s0__response {
  public:
     class s0__ArrayOfResultFile * files;
     class s0__Log * log;
};
```

Explanation: We can observe in line 1 that Element node is used to declare a new Version data type, which is equivalent to XSD string ('s' is the namespace prefix for XSD namespace). The C++/C equivalent for the Element node will be a typedef statement that will declare a new data type Version in the Definition namespace (s0) equivalent to XSD string.

In Line 2 Element node is used to declare an equivalent for response type in 's1' namespace. As the original data type is user defined a new data type is created in Definition namespace (s0) with the same members as the original data type.

4.1.5 Simple Types:

SimpleType nodes can be used to declare enumerations and enumeration masks. An enumeration mask is represented in WSDL as a SimpleType node containing a List node and a Restriction node with the base attribute set to XSD string. An enumeration is represented as a SimpleType node containing a restriction element with the base attribute set to XSD string.

The *values* attributes of Enumeration nodes represent the enumeration values for the data type. The data type name is give by the *name* attribute of SimpleType. The enumeration or enumeration mask id declared in the Definition name space for the document.

Example for Enumeration Mask:

1. <simpleType name="MASK">
2. <list>
3. <restriction base="xsd:string">
    a. <enumeration value="a"/>
    b. <enumeration value="b"/>
    c. <enumeration value="c"/>
4. </restriction>
5. </list>
6. </simpleType>

Example for Enumeration:

7. <simpleType name="ENUM">
8. <restriction base="xsd:string">
    a. <enumeration value="a"/>
    b. <enumeration value="b"/>
    c. <enumeration value="c"/>
9. </restriction>
10. </simpleType>

The C++/C equivalent for enumeration mask in gSOAP:

```
enum * ns__MASK { a, b, c };
```

The gSOAP compiler will interpret this as a special enum, where the enum values are powers of 2, i.e. a=1, b=2, c=4, to create a bitmask.

The C++/C equivalent for enumeration:

```
enum ns__ENUM { a, b, c };
```

Explanation: We can observe that SimpleType node in line 1 contains a List node and a Restriction node with its base attribute set to String, hence this represents an enumeration mask data type. The *name* attribute in line 1, which is MASK, gives the data type name. The Enumeration nodes at line 3.a to 3.c give the enumerations values. The new data type is declared in the Definition name space (ns) of the document.
The SimpleType node at line 7 represents an enumeration as it contains a restriction node with its *base* set to XSD String at line 8. The name is given by the *name* attribute at line 7. The enumeration values are given by the Enumeration node's value attribute at line 8.a to 8.c. The new data type is declared under the Definition name space (ns) of the SimpleType node.

4.2 Messages Equivalents:

Messages make up the input and output parameters for a function. A function may take any input consisting of any number of messages and may return an output of any number of messages. Messages are declared as Message nodes under the Definitions node. Each Message node consists of Part nodes, which are represented parts that make up the message.

Each part of message can either be basic or user defined.

The data type of Part can be specified using the *element* attribute or *type* attribute. If the *type* attribute is used, then the XML message equivalent of this part will contain only one node, if *element* attribute is used, then the XML message equivalent will consist of nodes corresponding to each member of user type. So while creating the total XML message equivalent we have to expand all the members of the user type as a part of the message.

If the message's part is specified by a *type* attribute, then the name of the message part is given by the *name* attribute of the Part node. In case *element* attribute is used, the *name* attribute is ignored, as the member variable names are used as the names of part.

In either case the C++ message equivalent will be a series of C++ data types qualified with namespaces. The namespaces are ignored if the part is of array type.

Example of Message consisting part nodes with type attribute:

1. <WSDL:message name="searchResponse">
    a. <WSDL:part name="keyword" type="SOAP-ENC:string"/>
    b. <WSDL:part name="product_group" type="SOAP-ENC:string"/>
    c. <WSDL:part name="product" type="md:ArrayOf_md_Product"/>
2. </WSDL:message>

C/C++ Equivalent

SOAP_enc__string _keyword, SOAP_enc__string _product_USCORE_group,

class ArrayOf_USCORE_md_USCORE_Product * _product;

Explanation: The message node in line 1 represents the message structure of a "searchResponse" message. The Part nodes in line 1.a to 1.c give the parts of message. The data type of each part is given by the *type* attribute. We can observe that the parts represented by line 1.a and l.b are soapencoding strings, which is a basic type and the part in line 1.c is an array. So the C/C++ equivalent of this message will be two soapencoding strings variables qualified by SOAP_ENC name space and an array_of_md_product variable without any namespace qualifier. The names of variables are given by the *name* attributes in line 1.a to 1.c.

An example of Message consisting of part nodes with type element:

1. <message name="GetEmailforDomainSOAPIn">
    a. <part name="parameters" element="s0:GetEmailforDomain" />
2. </message>

```
class s0__GetEmailforDomain {
  public:
      xsd__string  Query;
      xsd__string  LicenseKey;
};
```

The C/C++ Equivalent

```
xsd__string  Query, xsd__string  LicenseKey,
```

Explanation: The message node at line 1 gives the equivalent for "GetEmailforDomainSOAPIn" message because of the *name* attribute's value in Message node. This Message node consists of only one Part node, but the part uses an *element* attribute and its value is identified to be a user-defined type. So the C/C++ equivalent of the Part node in line 1.a will be the members of the GetEmailforDomain data type, which are 2 XSD strings variables. The variables names in the user data type are used as the names for parts and the *name* attribute in line 1.a is ignored.

4.3 Operations

Each Operation node gives a function provided by the web service. The Operations nodes are located under the Binding node. Each operation node provides the name, soap header information and soap action information for the operation. The soap action is a string that will be added to the soap message while sending a request. Each operation is defined in the Definition namespace of the document. The soap action information is used as a gSOAP compiler directive, printed before each operation declaration. The directive included the namespace for the function, the function name and the SOAP action string. If the SOAP action is not given for an operation, an empty string is used.

The soap action directive is given as

//gsoap <namespace> service method-action: <method Name> "soap action string"

Where <name space> is the namespace in which the operation is defined. <method name> is the name of operation without the name space prefix and soap action string is printed in quotes.

The operation name is given by the *name* attribute of the operation tag. The Input and Output nodes in the operation node give the type of encoding used for the input and output messages. If any operation uses a *literal* encoding, a compiler directive is generated specifying that the current namespace uses doc literal encoding.

The input and output arguments of an operation are given under the PortType node. The PortType node contains the Operation nodes corresponding to all the operations provided by the web services. Each Operation node can contain any number of Input and Output nodes. Each Input or Output nodes *message* attribute points to a message node. So the message equivalent of Input and Output nodes gives parameters for the operations.

According to gSOAP conventions an operations can have any number of input parameters but can have only one output parameter. So a new output data type is created

with all the variables of the output message. A pointer to this newly created data type is used as the output parameter. The new data type is named after the function name, by appending “Response” to the function name.

An example for Operation node:

1. <WSDL:operation name="browse">
    a. <WSDLSOAP:operation SOAPAction="http://majordojo.com/AmazonQuery#browse"/>
2. <WSDL:input>
    a. <WSDLSOAP:body encodingStyle="http://schemas.XMLSOAP.org/SOAP/encoding/" namespace="http://majordojo.com/AmazonQuery" use="encoded"/>
3. </WSDL:input>
4. <WSDL:output>
    a. <WSDLSOAP:body encodingStyle="http://schemas.XMLSOAP.org/SOAP/encoding/" namespace="http://majordojo.com/AmazonQuery" use="encoded"/>
5. </WSDL:output>
6. </WSDL:operation>

The operation node in PortType node corresponding to the above operation

7. <WSDL:operation name="browse">
    a. <WSDL:input message="md:browseRequest"/>
    b. <WSDL:output message="md:browseResponse"/>
8. </WSDL:operation>

C++/C equivalent:

9. //gsoap md service method-action: browse"http://majordojo.com/AmazonQuery#browse"
10. md__browse(SOAP_enc__string associates_USCORE_id, SOAP_enc__string product_USCORE_group, SOAP_enc__string category, md__browseResponse * out);

Explanation: The operation node at line 1 gives the information about the “browse” function declared in the Definition namespace (md). The *soapaction* attribute at line 1.a gives the action string, which results in the gSOAP compiler directive at line 9. We can observe that the operation is not using a literal encoding, so no literal directive is

required. The Input and Output nodes give the input and output arguments for the operation. A output data type is created by the name md_browseResponse as the operation name is “md_browse”, this data type contains the members of the Output message. If the operation call is successful, SOAP_OK is returned.

4.4 Web Service Attributes: Location, Name and Namespaces

Each web service is associated with a name and a location. The name of the web service is given by the *name* attribute of the Service node; the location (physical address) is given by the *location* attribute of the Address node.

A SOAP client needs the location of the web service to establish communication. The location is given in the header file as a gSOAP directive in the format below.

//gsoap <namespace> service location: <url>

<name space> is the name space in which the service is defined, this is nothing but the documents Definition namespace . The <url> is the actual location of the web service.

The name of service is also represented as a gSOAP directive in the header file. The name directive is given in the format below.

//gsoap <name space> service name: <service name>

Here <name space> is the name space pointed by the targetnamespace attribute and the <service name> is the name attribute of the service node.

The name space URI for the web service is also given using a gSOAP directive. The format is given below.

//gsoap <target name space> service namespace: <uri>

Where <target Name space> is the Definition name space for the document and <uri> is the URI equivalent for the name space.

1. <WSDL:service name="AmazonQuery">
2. <WSDL:port binding="md:AmazonQuerySOAPBinding" name="AmazonQuery">
3. <WSDLSOAP:address location="http://majordojo.com/cgi-bin/amazon_query.cgi"/>
4. </WSDL:port>
5. </WSDL:service>

Information is transformed to C/ C++ as:

//gsoap md service namespace: http://majordojo.com/AmazonQuery

//gsoap md service location: http://majordojo.com/cgi-bin/amazon_query.cgi

//gsoap md service name: SOAPAmazonQuery

4.5 Importing Documents

A WSDL file can import other file. This helps to reuse the commonly used data types of WSDL structures. Import nodes can be used in any part of the WSDL document. Importing a document has the same effect as inserting the imported document with the import node. To achieve this a search is performed over the document to locate all Import nodes. The document being imported is given by the *location* attribute of the import node. In case the *location* attribute is not present the import node is ignored. As the imported document may also contain more Import nodes, the process of importing need to be continued until no more documents are being imported or until no more Import nodes are found.

The imported document is inserted in place of an existing import node.

An example for Import node

<import location="http://www.whitemesa.com/interop/InteropTestC.WSDL"
namespace="http://SOAPinterop.org/"/>

This node will be replaced by the document present at

“http://www.whitemesa.com/interop/InteropTestC.WSDL”.

4.6 Multiple namespace prefix's pointing to the same URI

In many cases more one than one name space prefix is used to identify the same name space (uri). This leads to confusion in identifying the data types. To resolve this a global name space map is maintained between the URI and the name space prefix. Before identifying any data type, a search is performed in global name space map for the URI. If a match exists then the already used prefix is used for identifying the data type. If there is no match then, a new pair is added to the map. This pair resolves the conflicts, as there is a unique association exists between the URI and name space prefix in the header file.

```
1.  <schema
2.  XMLns="http://www.w3.org/2001/XMLSchema"
3.  targetNamespace="http://www.tempuri.org/n.xsd"
4.  XMLns:n="http://www.tempuri.org/n.xsd" >


5.  <complexType name="BaseRequest" abstract="true">
6.  <sequence>
        a. <element name="applicationID" nillable="true" type="xsd:string" />
7.  </sequence>
8.  </complexType>


9.  <complexType name="SendMelodyRequest">
10. <complexContent>
11. <extension base="newNs:BaseRequest" XMLns:newNs = ="http://www.tempuri.org/n.xsd" >
12. <sequence>
        a. <element name="melodyID" type="SOAP-ENC:string" />
13. </sequence>
14. </extension>
15. </complexContent>
16. </complexType>
```

In this case we can see at line 5 that BaseRequest is declared in the *n* namespace, which points to *"http://www.tempuri.org/n.xsd"*. But at line 11, the BaseRequest is referenced using *newNs,* where *newNs* points to *"http://www.tempuri.org/n.xsd"*. The Importer is

confused because of multiple associations of name spaces to same URI. This can be avoided by using a global namespace map. In this case when declaring BaseRequest a namespace URI pair corresponding to {n,*"http://www.tempuri.org/n.xsd"}* is added to the global name space map. Before using *newNS*, if we look into the global map, we can observe that a name space binding already exists for the URI. So we would use *n* instead of *newNS*, which resolves the confusion.

4.7 gSOAP Naming Rules

GSOAP uses certain rules to name the data types, member variables and operations. According to these rules all the keywords in c, c++ are appended with a '_'. So "result" which is a key word is converted to "result_" in the header file. If a variable already exists with the same name, then an extra '_' is added.

All '-' are converted to a '_'.

All '_' are converted to *_USCORE*.

All '.' are converted to *_DOT*.

The data type name and the name space of the data type is separated by '__'. Other special characters are represented by '_xHHHH', where 'H' is a hex digit to produce a two-byte Unicode character.

4.8 Data Type and Operation Generation

The Importer is designed to generate C and C++ header files. So we need to generate two different equivalents for each data type and operation. To avoid this all data types and operations are internally stored in a language independent form, which are interpreted only when writing the header file. This form will enable the Importer to generate header files in any language with little modification.

4.8.1 Internal Storage of Data Types and Header File Generation

All data types are stored in hash tables, where the key is the data type name and the value is actually a string containing the language-independent description of the data type. The template of any data type is :

<Key> = <dataType Number>\n <list>

<Key> will give the name space qualified name for the data type and <data type number> gives the category of data type but the key value is not in accordance to gSOAP conventions. The <list> part is interpreted based on the <data type number>.

The <data type Number > can take values from 1 – 6. This is designed in numeric format so that it can be extended in future.

1. **Enumeration Mask:**

If Data Type number is 1, then the data type is identified to be an enumeration mask. In this case the <list> gives the list of enumerations for the mask.

Eg.

```
enum * ns__MASK { a, b, c };
```

Will be internally stored as

“ns:MASK” “1\n a, b, c”

2. **Enumeration:**

If the data type number is 2, then the data type is identified to be of enumeration kind. In this case the <list> contains the enumeration values for the data type.

So consider the line:

<list> = <enumeration value>,<enumeration value>,<enumeration value>……

Here <enumeration value> represent the values enumerated by the data type.

This will be internally stored as

“ns:ENUM “ “2\na, b, c“

3. **Classes / Structure**

If the data type number is 3, then the data type is identified to be a class. In this case <list> contains member variables description separated by new line character.

So the

<list> = <member description>\n<member description>\n<member description>……

Each member variable description will be in the format below:

<data type>/t<is pointer>/t<variable name>/t< type>

We know that any member of a structure or class can be of a basic type, user defined type or an enumeration type. To represent all these types integer values are used.

<data Type> gives the data type of member. The name is not in accordance to the gSOAP conventions.

<is pointer> is set to 1 then the member is a pointer variable. If the value is set to 0 then the members is not a pointer.

<Variable Name> gives the exact name of variable. This name is according to the gSOAP conventions.

<type> takes an integer value ranging from 0 - 3.

I. XSD, SOAP-ENC

If the value in type is 0, the member is variable of basic type, which can be XSD or SOAP-ENC type.

II. Classes

If the value in <type> is 1, the member is a variable of user-defined type. Arrays also come in this category.

III. Enumerations, Enumerations mask

If the value in <type> is 2, the member is a variable of enumeration or enumerations

IV. Reserved

If the value in <type> is 3, the member is of reserved type. This is used when gSOAP rules are not to be applied on the data type and variable names. Same in case of integer members.

4. **Arrays.**

If the data type number is 4, then the data type is identified to be an array. Arrays are also stored similar to the classes. The only difference between the Class and array is that array will have only 2 members in <list>.

<member description>\n
<member description> \n(for __size varible)
<member description> (for __offset varible)

5. **Extended Structures.**

If the data type number is 5, then the data type is identified to be an extended Class. In this case <list> will be interpreted as

<base class name>\t<number members of base class>
<member description>
<member description>
<base class name> gives the base class for the current class. This is used to know the extended class name in C++.
<number of members in base class> this gives the number of members in the base class let it be N.

The first N <member description>actually corresponds to the base class. The rest of them will be the members corresponding to the current data type.

6. **Typedefs.**

If the data type number is 6, then the data type is identified to be a typedef statement. In this case <list> will be interpreted as a single <member description>. The <variable name> and <is pointer> fields of the description are of no importance.

4.8.2 Internal Storage of Operation information

Operations are stored in hash tables, with the operation name as key and the description as the value. The key stored is not in accordance to the gSOAP standards**.**

<operation name> : <description>

The <description> is a collection of members that make the input /output parameters. Each parameter storage is similar to a class member's storage. So the <description> can be given as

<member description>\n
<member description>\n
<member description>\n
<member description>\n
<member description>\n
…….

4.9 Node Search

Node Search is done using three classes NodeSearch, NSNode and NSNodeSearch. Node Search class recursively finds the node using a depth first algorithm. This class it ignores all namespaces bindings associated with the nodes and finds the nodes only by their names. NSNode is a wrapper class over the traditional Node class of java. Each NSNode

object contains a hash table with the namespace mappings and the assigned URI's. Namespace mappings are very handy in identifying the XSD and WSDL namespaces. NSNodeSearch class searches the node using a depth first algorithm but remembers all the namespaces encountered through the search. NSNodeSearch class search returns a NSNode object.

5. How to use the WSDL Importer

Creation of a valid gSOAP client header file from the given Web service description language document is a two-step process.

Step 1: Validate the WSDL document

The WSDLCPP compiler doesn't incorporate extensive error checking of the WSDL document. To make up for this the input WSDL document has to be passed through a WSDL validation tool when available. This will prevent many errors that may occur during header file generation.

Example: In case of stock quote web service the WSDL file is located at http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl. We can use any of the free online tools available. Cape clear has a very handy tool to validate the WSDL files.

Step 2: Compile the WSDL document

Pass the document through the WSDL compiler, which generates the gSOAP header file and a sample gSOAP client.

The command to invoke the compiler is

"Java wsdlcpp <filename>.wsdl <-c>"

The compiler expects the WSDL file to end with ".wsdl" extension. By default the client and the header file is generated in C++. A C header file can be generated by including a "-c" option as an argument.

If the WSDL file is compiled successfully, then the compiler prints the message:

Processing Done.

Header File <filename>.h

Sample C File <filename>.c

<filename>.h and <filename>.c are created only if the compilation is successful. In case of any error a error message is displayed. The error can be either due to some unsupported feature in WSDLCPP or due to some problem in the WSDL file.

Example: In case of stockquote example copy the WSDL file to a local directory and run the compiler.

Java wsdlcpp stockquote.wsdl

This generates 2 files stockquote.h and stockquote.c

StockQuote.h :

```
//gsoap soapenc schema namespace: http://schemas.xmlsoap.org/soap/encoding/
//gsoap s schema namespace: http://www.w3.org/2001/XMLSchema
//gsoap http schema namespace: http://schemas.xmlsoap.org/wsdl/http/
//gsoap tm schema namespace: http://microsoft.com/wsdl/mime/textMatching/
//gsoap soap schema namespace: http://schemas.xmlsoap.org/wsdl/soap/
//gsoap mime schema namespace: http://schemas.xmlsoap.org/wsdl/mime/
//gsoap s0 schema namespace: http://www.webserviceX.NET/

//gsoap s0 service namespace: http://www.webserviceX.NET/

//gsoap s0 service location: http://www.webservicex.net/stockquote.asmx
//gsoap s0 service name: soapStockQuote

/*start primitive data types*/
typedef char * xsd__string;

/*end primitive data types*/
```

```
typedef xsd__string  s0__string ;

class s0__GetQuoteResponse_ {
  public:
      xsd__string  GetQuoteResult;
};

class s0__GetQuote {
  public:
      xsd__string  symbol;
};

class s0__GetQuoteResponse {
  public:
      xsd__string  GetQuoteResult;
};

//gsoap s0 service method-action: GetQuote_ "http://www.webserviceX.NET/GetQuote"
s0__GetQuote_( xsd__string  symbol, s0__GetQuoteResponse_ * out );
```

StockQuote.c

```
#include "soapH.h"
#include "soapStockQuote.nsmap"
main()
{
      struct soap soap;
      soap_init(&soap);
          if (soap_call_s0__GetQuote_ ( &soap,
          "http://www.webservicex.net/stockquote.asmx",
          "http://www.webserviceX.NET/GetQuote",/* xsd__strin
          g  symbol, s0__GetQuoteResponse_ * out*/ ))
              soap_print_fault(&soap,stderr);
}
```

In this case the stockquote.h contains the header file information to access the web service and stockquote.c gives the framework of the sample client.

Step 3: Compile the header file to generate the necessary GSOAP helper file.
Example: Soapcpp2 stockquote.h
Will generate the necessary marshalling and un marshalling routines needed. The main files generated are soapC.cpp and soapClient.cpp.

Step 4: Edit the sample client.
Modify the sample client's content to the actual values needed and incorporate the business logic needed.

Example: The modified stock quote client to accept the symbol as a command line argument and to display the stock details is given below

StockQuote.c

```
#include "soapH.h"
#include "soapStockQuote.nsmap"
main(int argc,char *argv[])
{
      struct soap soap;
      soap_init(&soap);
      s0__GetQuoteResponse_  out;

      if(argc<2)
      {
         Printf("Usage stockquote <symbol>")
         Exit(0);
      }

      if (soap_call_s0__GetQuote_ ( &soap,
      "http://www.webservicex.net/stockquote.asmx",
      "http://www.webserviceX.NET/GetQuote","IBM",&out ))
      {
            soap_print_fault(&soap,stderr);
           exit(1);
         }
      printf("Stock Details of %s :\n\t = %s\n",argv[1],out.GetQuoteResult);
}
```

Sample run of the stockquote example :

Run the quote program for Citrix Systems:

```
kiran@diablo:~/gsoap-sun8-2.6>stockquote CTXS
Stock Details of CTXS :
      =
<StockQuotes><Stock><Symbol>IBM</Symbol><Last>92.28</Last><Date>4/16/2004</Date><Time>4:
```

00pm</Time><Change>-
1.69</Change><Open>92.30</Open><High>92.35</High><Low>91.04</Low><Volume>11118100</Vo
lume><MktCap>156.8B</MktCap><PreviousClose>93.97</PreviousClose><PercentageChange>-
1.80%</PercentageChange><AnnRange>78.16 - 100.43</AnnRange><Earns>4.48</Earns><P-
E>20.98</P-E><Name>INTL BUS MACHINE</Name></Stock></StockQuotes>

6. References:

[1] W3C Candidate Recommendation 19 December 2002, "SOAP Version 1.2 Part 0: Primer", Available, http://www.w3.org/TR/2002/CR-SOAP12-part0-20021219/

[2] IEEE CCGrid Conference 2002, Robert A. van Engelen and Kyle A. Gallivan, "The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks"

[3] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, 2001, Web Services Description Language (WSDL) 1.1, Available, http://www.w3.org/TR/2001/NOTE-WSDL-20010315

[4] W3C, W3C Architecture Domain, 2002, "Extensible Markup Language (XML)", Available, http://www.w3.org/XML/

[5] IDG Books Worldwide, Elliotte Rusty Harold, "XML Bible", First Edition, 2000

[6] World Wide Web Consortium, "Namespaces in XML" Available, http://www.w3.org/TR/REC-xml-names/

[7] W3C Recommendation , 1998 "Document Object Model (DOM) Level 1 Specification" http://www.w3.org/TR/REC-DOM-Level-1/introduction.html

[8] Dave Winer, 1999 "XML-RPC Specification" Available, http://www.xml-rpc.org/spec

[9] Microsoft, Web Services Specifications Available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsrvspec/html/wsspecsover.aspList

[10] Robert van Engelen, Gunjan Gupta, and Saurabh Pant, Developing Web Services for C and C++, in IEEE Internet Computing Journal, March, 2003.

[11] Robert van Engelen, Code Generation Techniques for Developing Web Services for Embedded Devices, in the proceedings of the 9th ACM Symposium on Applied Computing SAC, Nicosia, Cyprus, 2004Dave Winer, 1999 "XML-RPC Specification" Available, http://www.xml-rpc.org/spec