THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

DISTRIBUTED FILE ACCESS WITH WEB-SERVICES

By

SMITA S. KULKARNI

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 Smita S. Kulkarni defended on Tuesday, April 27, 2004.

Dr. Robert van Engelen
Supervising Professor

Dr. Lois Wright Hawkes
Committee Member

Dr. Kyle Gallivan
Committee Member

To, Dear Aai Baba

Acknowledgements

My deepest thanks to my major professor Dr. Robert van Engelen, for his original idea and the design of the project. I learned a lot from working with him on this project. I want to thank Dr. Lois Hawkes and Dr. Kyle Gallivan for agreeing to be on my Master's Project defense committee. Finally I want to thank my parents who are constant source of inspiration and support.

Table of Contents

Abstract

This project report describes an efficient distributed file access system with a standardized Web services interface developed with the gSOAP toolkit. The file access system built with gSOAP can transmit binary data efficiently using multimedia-streaming techniques. With multimedia streaming, binary data is retrieved from data sources at run time in parts without the need to store or buffer the entire data contents. Likewise, the data is streamed to the designated data sinks at the receiving side. Streaming is implemented in gSOAP with function callbacks to handle the data streams to and from the application and to encode them as attachments to the XML-based Web services messages. There are a number of advantages of using this approach. The standardized Web services interface uses SOAP/XML messaging over HTTP as the transport mechanism. HTTP can pass through most firewalls and supports SSL encryption and compression. Because SOAP (Simple Object Access Protocol) is an XML-based message format, SOAP/XML messaging provides a non-proprietary open standard for interoperability between diverse platforms and disparate organizations. SOAP uses XML message format and therefore SOAP has inherited all the advantages that XML provides. The resulting system based on SOAP with DIME (Direct Internet Message Encapsulation) implements an efficient binary-based data transfer mechanism with a standardized open Web services interface for distributed file access.

1. Introduction

1.1 Background Information

The most basic data transfer scenario involves transfer of data from one point to other. There are different mechanisms that are provided for transferring data from one application to another. Although this seems to be an easy task, the mechanism has to address different issues before the actual data transfer begins. The two applications might have different file naming convention or different ways to represent the data or they could have different directory structures. Therefore a mechanism is needed which will address all such problems that can occur in data transfer. Several protocols and mechanisms have been proposed and developed for an inter application data exchange. The brief review of few such mechanism is given below.

Sun Microsystems's Remote Procedure Call:

RPC compiler generates code that is used by client and server to exchange data over the network. RPC data is encoded using the e**X**ternal **D**ata **R**epresentation. XDR defines how integers, floating-point parameters and strings etc are defined for network transmission. This is called as parameter marshaling. However it does not support pointer based data structures such as graphs. RPC provides a mechanism for authenticating procedure calls from one machine to another.

File Transfer Protocol:

FTP is a standard mechanism provided by TCP/IP for downloading or uploading files from one computer to another over the network. This is the most popular protocol that is used over the web for file transfer. FTP establishes two connections with the host, one for the data transfer and one for the control information such as commands and responses transfer. FTP uses two well-known TCP ports: port 20 for the data transfer and port 21 for control information transfer. The FTP Server allows the authorized users to upload and download files from it. The files are stored on the FTP server. The FTP client is a program that makes it possible to upload and download the files from the FTP server. If the user is not an authorized user then with the help of anonymous FTP user can connect to FTP site for file transfer if it is allowed.

Common Request Broker Architecture:

CORBA is a specification that allows programs that are created on different platforms by different vendors to communicate over the network. It's an open standard for working with distributed objects. CORBA used **I**nternet **I**nter **O**RB **P**rotocol for data transmission between the CORBA applicants.

Microsoft Distributed Component Object Model:

DCOM is a protocol that allows the client to directly communicate with each other in a secure and efficient way. Even if DCOM is a platform independent protocol it is mainly used within the windows environment.

Sun Microsystems's Java Remote Method Invocation:

JAVA RMI helps user to create distributed Java technology based applications enabling java remote methods, which are on different hosts. The marshaling of objects for the transmission is done automatically. Even the sterilization of complex data structures like arrays, lists, graphs, and maps is supported.

NFS

Network **F**ile **S**ystems connect virtual and physical file systems over a LAN. Mounting a file-based drive is simple with OS support and the OS typically provides a transparent file access system to the users. NFS assumes a high level of reliability and security of the LAN.

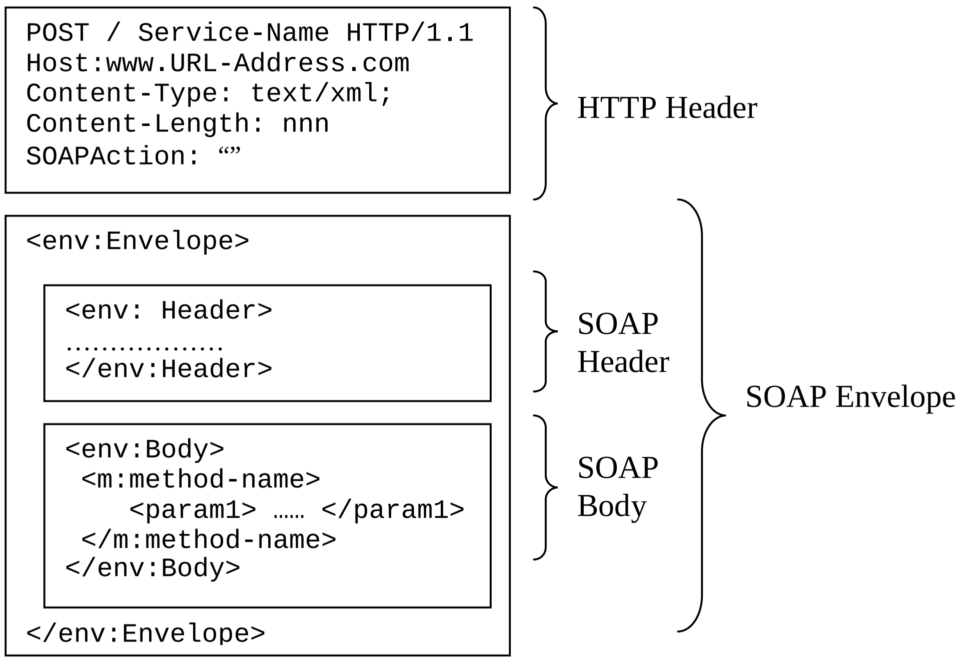
1.2 SOAP

"**S**imple **O**bject **A**ccess **P**rotocol (SOAP) is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses. SOAP can potentially be used in combination with a variety of other protocols." [W3c SOAP 1.1 specification] Though SOAP can be used with variety of other protocols, but this document describes SOAP implementation with HTTP protocol because of few advantages of HTTP such as it is available on all the platforms, it requires very little runtime support and it is firewall friendly.

Figure-1 describes the SOAP message structure[8]. A message consists of an envelope root element, an optional SOAP header element to encode the meta-information such as authentication information, signatures, message routing information etc, encoding rules describing how messages should be processed and SOAP message body which describes the remote procedure call representation that is RPC responses and parameters. Clients can send complex data types such as structures or arrays as the RPC parameters. The value of each parameter appears as the XML element within the message body. To carry the error information, SOAP Fault Element is used.

Example of SOAP Request and Response:

Figure-2 and Figure-3 shows an example of a sample SOAP/HTTP request and SOAP/HTTP response. The SOAP/HTTP request contains a block called GetTodaysWeather that takes a single parameter, name of the city. The service's response to this request contains a single parameter, the temperature of the city in degree Fahrenheit.



```
POST / Service-Name HTTP/1.1
Host:www.URL-Address.com
Content-Type: text/xml;
Content-Length: nnn
SOAPAction: “”

<env:Envelope>

  <env: Header>
  ………………
  </env:Header>

  <env:Body>
   <m:method-name>
      <param1> …… </param1>
   </m:method-name>
  </env:Body>

</env:Envelope>
```

Fig: 1- SOAP Message Structure

```
     POST / GetWeather HTTP/1.1
     Host: www.TallahasseeLocalWeather.com
     Content-Type: text/xml; Charset = “utf-8”
     Content-Length: nnn
     SOAPAction: “”

     <env:Envelope xmlns:env =
            “http://schemas.xmlsoap.org/soap/envelope ” >
      <env:Body>
         <m:GetTodaysWeather
            env:encodingStyle=
            http://schemas.xmlsoap.org/soap/encoding”
            xmlns:m = “”>
            <City> Tallahassee </City>
         </m:GetTodaysWeather >
        </env:Body>
       </env:Envelope>
```

Fig: 2 - SOAP HTTP Request

```
HTTP/1.1 200 OK
Content-Type: text/html; Charset= “utf-8”
Content-Length: nnnn

<env:Envelope xmlns:env=
      “http://schemas.xmlsoap.org/soap/envelope” >
 <env:Body>
  <m:GetTodaysWeatherResponse
      env:encodingStyle=
           http://schemas.xmlsoap.org/soap/encoding”
           xmlns:m = “”>
    <Temperature> 55 </Temperature>
  </m:GetTodaysWeatherResponse>
 </env:Body>
</env:Envelope>
```

Fig: 3 - SOAP HTTP Response

Why SOAP:

SOAP is used for the application because of some advantages of SOAP, which are as follows:

1. **Interoperability:** With the use of technologies such as XML and HTTP SOAP can be used within the applications that run on different operating systems. That means the client written on Microsoft platform can invoke methods from the application running on Linux.
2. **Open Standard:** SOAP is built upon the technologies such as XML and different transportation protocols such as HTTP, FTP that are not vendor specific. Therefore it’s been uniformly accepted by the industry.
3. **Firewall Friendly:** When SOAP is used with the HTTP protocol, the SOAP message packets can easily bypass the firewall as it uses the standard port 80 for the message data transmission. However it is possible for the system administrator to track the SOAP specific HTTP headers to re-configure the firewall to block it.
4. **Easy maintenance:** The SOAP based system is easy to deploy. It requires minimum amount of setup including enabling the port for data transmission.

1.3 DIME

Why DIME:

The biggest strength of SOAP is that it can encapsulate XML data from any XML schema within the message. Still at many times there is a situation where the data is not in XML format. In that case if the SOAP is used for the data transmission then the data marshalling to and from XML becomes necessary. However this is not always an easy task as there are situations where this data conversion to and from XML is very expensive. e.g. Suppose the JPEG image file is to be transmitted. The JPEG format for holding the image is highly structured format. If this image data is to be converted to XML schema then the conversion of the data to and from XML format can cause tremendous slow down because of the large size of the image. The other situation occurs when the data that is to be transmitted between the two systems, which do not use XML. In that case the data is required to convert to and from XML. In such a situations the data is serialized and deserialized from the original format to XML only for the sake of data transmission. Therefore a mechanism is needed that will allow SOAP to transmit the binary data with itself. This mechanism is provided by DIME. Hence it is used for this application.

Direct **I**nternet **M**essage **E**ncapsulation (DIME) is a specification that allows multiple binary records to be included within it. With DIME there is no restriction on size and format of data. The data could be of any type i.e. either a JPEG or GIF file or a SOAP message or binary digitally signed data. Also while transmitting the data it is not necessary to find out the total length of data.

DIME Structure:

DIME message is organized into different records. There is no restriction on the number of records. Each record has record header and data. The first record sets the message begin flag set and the last record sets the message end flag. Because of these flags it is not necessary for the application to know the data size in advance. In addition to these flags the header contains few more flags. The length of each record is different as the length of data within each record is different. The sequence of the data records must be maintained while data transmission.

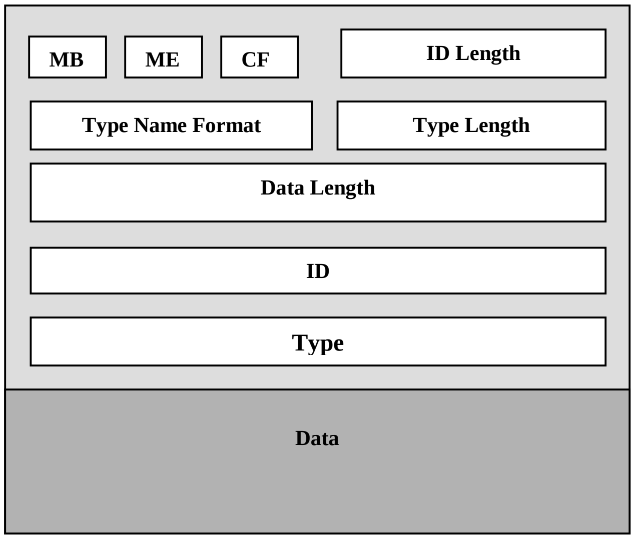


Fig: 4 - DIME Data Record Format

Fig: 4 shows the DIME data record format[2]. The light gray portion of the record specifies the record header and the dark gray portion specifies the data within the DIME. The first three bits are three flags. MB is the message begin flag, ME is message end flag and CF is Chunked flag. The next field is ID length, which length of identifier stored in field ID, which is used to identify a particular record in the DIME message. The type name format specifies the mechanism used to describe the data type and the type length field specifies the length of the type field. The data length field specifies the length of the data within the record. The data length field is of 32-bit size, which specifies the maximum data size of 4GB.

Need for data chunking:

To avoid the size limitation on the data DIME offers a very good solution called data chunking. There is need for data chunking for various reasons like to allow the application to send the data larger than 4-gigabite of size. Also even if the application needs to send the data of lesser size, it’s not always possible for the application to allocate a single buffer to store the data before data transmission. Therefore the chunking option is needed with which the part of data can be read at a time and sent over. The other reason is sometimes the application is unaware of the size of data to be transmitted e.g. output of the Database Query, in such a case data chunking provides the good solution. Whenever the application need to start chunking simply turn on the CF flag in the DIME header and when the complete data is sent turn off the CF flag. The ID and type field for all the records for which CF flag is set, is same.

Example:

Consider a scenario where a server needs to send a JPEG file to a client. The file is of huge size. Therefore sending the whole file as a single attachment is difficult for the server and reception of such a huge data into single buffer will be difficult for the client. Therefore the concept of 'data streaming' comes into the picture. In such a case the server can send the file as a DIME attachment with the option of data chunking.

DIME vs. MIME:

Multipurpose **I**nternet **M**ail **E**xtension provides a mechanism with which various types of complex files can be sent as an attachment with the email messages. It works with any transport system that is compliant with SMTP. MIME does this task of simplifying and rebuilding of complex files by encoding a file and transporting it as a message body. A MIME-compliant user agent (UA) on the receiving end can decode the message to get the original file contents.

This looks very similar to DIME. But DIME provides certain advantages over MIME. DIME provides simplicity where MIME provides flexibility.

DIME specifies the data record by specifying the data record length in the header while MIME represents its data records by bounding the data with a unique string, called the MIME boundary. This string is defined at the beginning and at the end of the MIME message. The application has to scan the entire data in the message until it reaches the next string where it can determine the data record boundary. There is problem with this approach. To send an arbitrary block of data, the data must be persistent or buffered to determine a unique boundary string that does not match any part of the data to be transmitted. This prevents streaming, where a data stream does not require persistence or buffering of the entire data content. In MIME, any prior chosen separator string can occur within the data of the data record. In that case the MIME message format is broken.

With DIME, the application only needs to send the data in chunks, where each DIME chunk contains the relative local size of the data chunk. Memory allocation with DIME is much more straightforward as it specifies the size of the data record. With MIME it does not specify the size of data so that at the receiving side, the application has to incrementally enlarge the data space to allocate the memory for the receiving buffer.

MIME encodes data before sending it whereas DIME does not require encoding of binary data. DIME has simpler syntax. Content type specification for MIME is always "text/html", Whereas DIME supports content type specified with URI as well.

1.4 Architecture of a web-service

Before getting into the SOAP-DIME web service details, let's first study what is the architecture of a web service[8]. The figure-5 shows the layered architecture of a web service. The information and function description of each layer is given below.

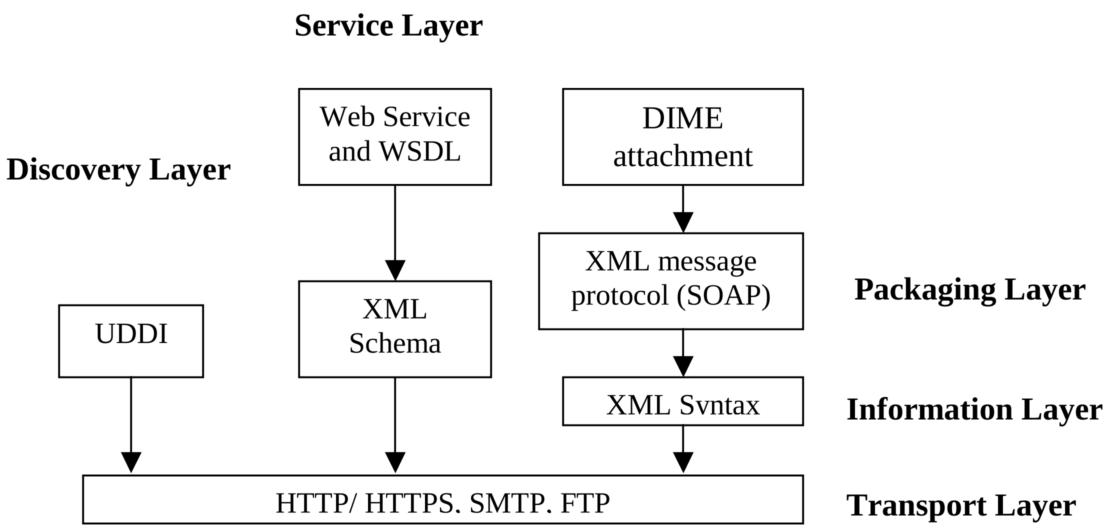


Fig: 5 – Layered Architecture of a Web Service

Discovery Layer: The discovery layer offers a way to know about the different web service providers. **U**niversal **D**escription **D**iscovery **I**dentification provides a mechanism for clients to dynamically find other web services. UDDI has 2 types of clients, one who want to publish the information about the service that it provides and the other client who wants to obtain that service.

Service Layer: Web **S**ervice **D**escription **L**anguage describes the SOAP/XML web service. WSDL provides the basic format of the service request over the different protocols and with the different encodings. It describes where the web service is, what function it can do and how the client can invoke it. WSDL also gives information about the communication protocol used to send message to the web service along with specific mechanisms like commands, headers or error codes.

Packaging layer: Packaging layer uses XML based SOAP protocol. Figure: 1 described in the Chapter-1 gives the detailed information about the SOAP message structure

Information Layer: Information layer carries the XML formatted SOAP message. The process of wrapping application data into XML format is called as XML Serialization which involves the task of XML encoding before sending the data and XML decoding after reception of the data.

Transport Layer: Firewall friendly HTTP or HTTPS are used to invoke web services with HTTP post request- response message exchanges.

1.5 RPC

Remote **P**rocedure **C**all is a programming model that allows the developers to work with method calls. RPC uses request/response type message exchange pattern. SOAP defines a standard way to map RPC calls. So that it is easy to translate between the method invocation and the SOAP call at runtime. To make the PRC call-using SOAP the application (in this case the client) needs the following things:

1. Endpoint URL: Where the server is located which will serve the client.
2. Method name: The different methods written by the server to serve the client.
3. Parameter names/ values: The parameter passed by the clients to the method located at server machine. This includes the return parameter as well.
4. Optional method signature
5. Optional header data

The information is conveyed to the server with WSDL files. First it maps the method signature to simple request / response structure, which later on has to be encoded in the XML format. According to the SOAP specifications the RPC request and response should be modeled as structures. The request structure should be named after the method. This structure contains all the input parameters, which are defined in the method signature. Even the order of the parameters should be the same. For the response structure the name of the response should be name of the method followed by the word 'Response'. This structure contains the return parameter and optional in-out parameters. The parameters passed to the method could be in any format like they could be arrays or integers or strings. To map these data structures to XML, SOAP defines the set of encoding rules. These rules explain how to map the data structures like arrays, structures strings to XML. RPC produces a SOAP fault in case of the error otherwise the method returns with the return parameter.

The figure-6 shows the RPC request / response message exchange[8]. The client's RPC stub routine marshals the function parameter in XML, wraps it in SOAP envelope and sends it over the network. At the reception side, the server RPC skeleton demarshals the function parameters sends the request to server and then sends the response from the server in the similar manner.

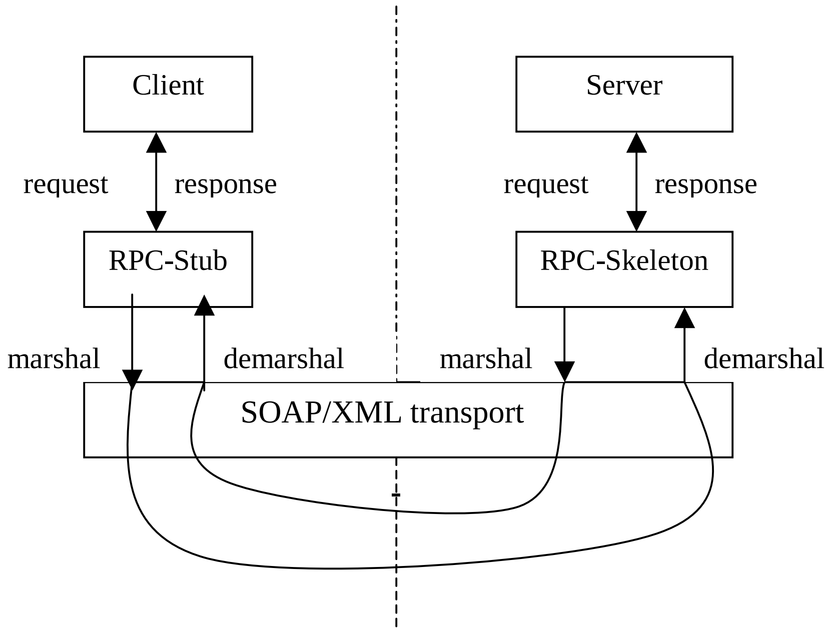


Fig: 6 – Remote Procedure Calling

1.6 gSOAP

gSOAP toolkit is a platform independent development environment for C and C++ web services. The main features of gSOAP are easy to use and good performance. It includes HTTP1.0/1.1 web server, XML parser/generator, RPC compiler and WSDL importer. This toolkit offers an easy to use RPC compiler that generates the stub and skeleton routines to integrate C and C++ applications into SOAP/XML web services. It automatically maps the C and C++ data types to equivalent XML data types. It generates the code to limit the application code size, which is very good for embedded systems. The gSOAP is compliant with WDSL and SOAP latest versions. It also supports a basic set of web service protocols.

Some of the highlights of gSOAP that are used in this application development are listed below:

- gSOAP is easy to use and has good performance
- gSOAP is the only toolkit that supports streaming DIME attachment transfers.
- gSOAP supports Zlib deflate and gzip compression (for HTTP, TCP/IP, and file storage).
- gSOAP supports SSL (for HTTPS).
- gSOAP supports HTTP/1.1 keep-alive, chunking, and basic authentication.

2. Design Issues

2.1 Project Outline

For this project, I have developed a Simple Image Server-using DIME. This server runs as CGI (not multi-threaded) or as a multi-threaded stand-alone web service. The server stores all the JPEG files, which are uploaded by different clients. For the security reason the client can access only the JPEG files that are in current working directory of the server. For this image server, I have developed DIME clients, which sends the request to the web service to upload or download or delete the files.

2.2 SOAP-DIME Web Service for Distributed File Access

Figure-7 shows the development and deployment stages of the gSOAP web service[8]. The application implements a set of SOAP compliant RPC functions to expose the service on the web. In this case there are four RPCs, first is to download data from the web service (gdx__get), second is to upload data on the web service (gdx__put), third is to view the list of data files on the server (gdx__list) and fourth is to delete files on the server (gdx__delete). The prefix gdx stands for "*gSOAP data access*". If the remote method uses compound data types such as arrays or structures etc then it is required to declare them in a header file. For this application all the structure declaration as well as the method prototype is in 'gdx.h'. This service can also be specified with a WSDL file, which can be preprocessed by gSOAP compiler to generate a header file. The gSOAP compiler generates the skeleton routine for each of the remote method described above and the serialization code for the data types. The skeleton routine can be used to implement the remote method in the SOAP web-service. These codes are compiled and linked with the service application to expose the service to the web.

The gSOAP RPC compiler generates a WSDL document describing the services that it provides in detail such as remote method information, types of parameters. In this case the WSDL document name is '*gdx.wsdl*'. This document can be registered and used by a developer of a client application to implement a procedure to invoke the web service.

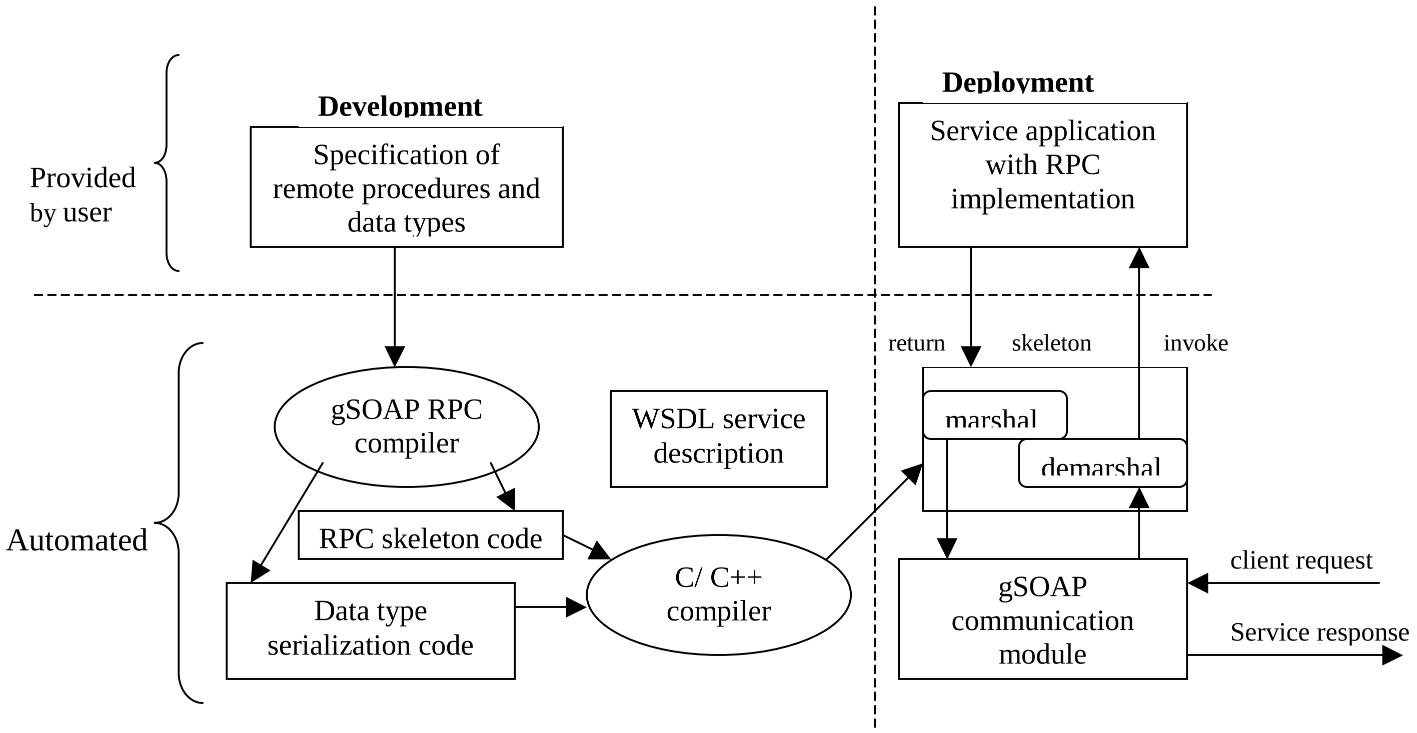


Fig: 7 – Development and deployment of web service

2.3 SOAP-DIME Client Application

The SOAP client developed has four remote methods with which it can either upload the file on the server or download the file from the server or can delete the file or get the list of available files. To invoke these remote methods the client needs the four stub routines for the remote operations. Figure –8 shows the development and deployment stages of a gSOAP client[8]. The input to gSOAP compiler is the header file (*gdx.h*), which is then processed by the gSOAP compiler to generate the stub routine for the client application. The primary stub's responsibility is to marshal the input data send the request to the designed SOAP service (server) over the network, wait for the response and once the response arrives demarshal the output data. The client application invokes the remote methods in exactly similar way as it invokes any local method. While downloading the file, the DIME client demonstrates the new DIME streaming feature.

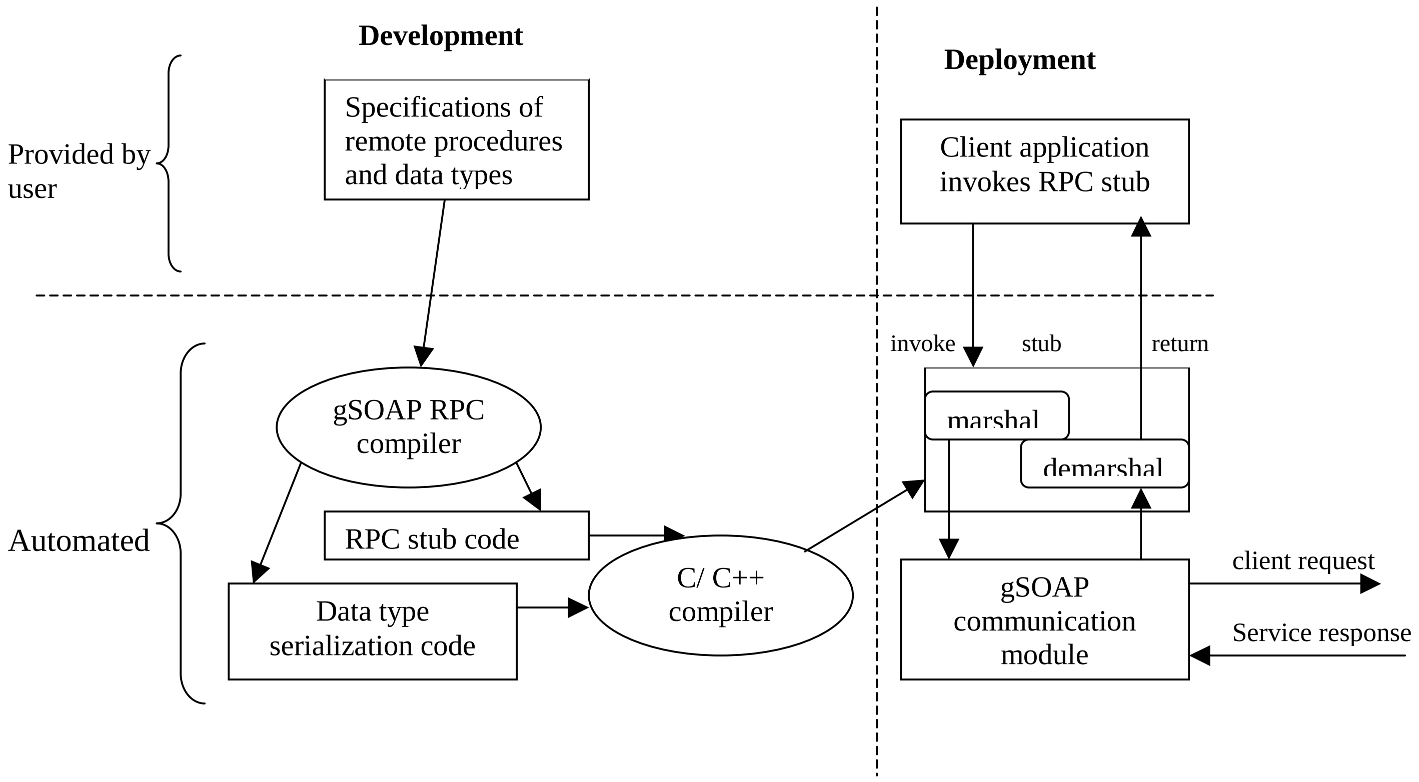


Fig: 8 – Development and deployment of a client

The detailed information about each method is as follows-

Downloading the file:

The remote method (gdx__get) specifies all the parameters needed to download the file from the SOAP web service. With the help of the function prototype the gSOAP compiler generates the stub routine to interact with the SOAP service. The remote method takes user authentication information and the resources information, which includes the list of names of files to be downloaded from the server as input parameters. Both these parameters are passed by value. According to gSOAP convention, the last parameter of the remote method is output parameter, which is passed by reference. Here the output parameter specifies the data of all the requested files. The implementation of this method is discussed in detail in the next section. The function prototype associated with this method is **int** whose value indicates whether the request is successfully processed or not. The output can be received as DIME attachment or as DIME streaming. Both these approaches are described in section 2.3 and 2.4 respectively. The client can change the name of the files, which he/she is downloading. This is an optional parameter.

Uploading the file:

The remote method (gdx__put) specifies all the parameters needed to upload the file on the SOAP web service. The stub routine, generated by the gSOAP compiler, interacts with the SOAP service. This remote method takes user authentication information such as username and password and list of files to be uploaded on the server as the input parameters. All the input parameters are passed by value. Every remote method needs to have an output parameter. Here the output parameter is the null parameter. The function prototype associated with this method is **int** whose value indicates whether the request is successfully processed or not. The files can be uploaded on the SOAP web service as DIME attachments or as DIME data streaming. Both these approaches are discussed in sections 2.3 and 2.4 respectively.

Deleting the file:

The remote method (gdx__delete) is written to delete the files on the SOAP web service. The stub routine, generated by the gSOAP compiler, interacts with the SOAP service. This remote method takes user authentication information such as username and password and list of files to be deleted from the server as the input parameters. All the input parameters are passed by value. The output parameter is the null parameter. With this option any client can delete the file on the server. With the help of authentication information the access to the files on the SOAP web service for the purpose of deletion can be restricted. This issue is discussed in the section-4, Future Work. The function prototype associated with this method is **int** whose value indicates whether the request is successfully processed or not.

Listing the files:

The remote method (gdx__list) specifies all the parameters needed to list all the files on the SOAP web service. For the security reason the method lists all the files in the current working directory of the SOAP web service. The stub routine, generated by the gSOAP compiler, interacts with the SOAP service. This remote method takes user authentication information such as username and password as the input parameter. The output parameter is the list of all the files in the current working directory of the SOAP web service. The function prototype associated with this method is **int** whose value indicates whether the request is successfully processed or not.

2.4 DIME File Attachment

gSOAP can transmit binary data with DIME with or without streaming. Without streaming the data is stored in XSD:base64Binary class. The structure is declared as follows –

```
struct xsd__base64Binary
{ unsigned char *__ptr;   // point to binary data block
  int __size;            // size of block
  char *id;             // optional DIME attachment ID
  char *type;          // DIME attachment type (MIME type)
  char *options;      // Additional information with DIME
};
```

The last three parameters of this structure are important to support DIME. gSOAP will test for the presence of the DIME-specific attributes at run time and use SOAP in DIME accordingly. The order of these fields is important. Any additional declarations can appear after the declaration of these fields. When the id or type field is non-NULL, DIME attachment transmission is used for the entire SOAP message.

2.5 DIME Data Streaming

Streaming DIME is achieved with callback functions to fetch and store data during transmission. For DIME output streaming, three function calls are available with the gSOAP tool and for DIME input streaming three functions calls are available[7].

DIME receiving-side streaming:

The three function calls that are available are:

1. *soap.fdimewriteopen( ):* Called by the gSOAP DIME attachment receiver at run time to start writing an inbound DIME attachment in the application's data store. The contents are stored in the applications data store with the multiple fdimewrite()

function calls, which are discussed below. The prototype for this function is as follows:

```
soap.fdimewriteopen = dime_write_open;
static void *dime_write_open(struct soap *soap, const char *id,
const char *type, const char *options)
```

The id field contains the image id, type field contains the image type i.e. “image/jpeg” and options field contains additional information about DIME attachment. In this case it contains the image name. The callback returns the handle of the file that is to be transmitted via DIME streaming. This handle is next passed to soap.fdimewrite() and soap.fdimewriteclose() functions.

2. *soap.fdimewrite():* Called by the gSOAP DIME attachment receiver at run time to write an inbound DIME attachment in the application’s data store in parts. The prototype for this function is as follows:

```
soap.fdimewrite = dime_write;
static int dime_write (struct soap *soap, void * handle, const char *buf, size_t len)
```

The handle field contains the file handle returned by the fdimewriteopen() callback. The buffer contains the inbound data of length len. In case of any error, the call back returns gSOAP error code otherwise the call back returns SOAP_OK. There could be multiple fdimewrite() calls.

3. *soap.fdimewriteclose():* Called by the gSOAP DIME attachment receiver at run time to close the data store. It is called either after successfully receiving the data or after any error occurrence.

```
soap.fdimewriteclose = dime_write_close;
static void dime_write_close(struct soap *soap, void *handle)
```

The handle contains the file handle returned by the fdimewriteopen() callback.

In the SOAP structure, there is one field called “user” field, which is available to pass the user-defined data. For this application this field is used to pass the pointer of the list of image file names. The detailed information is given in the next section.
The following diagram explains the DIME receive streaming –

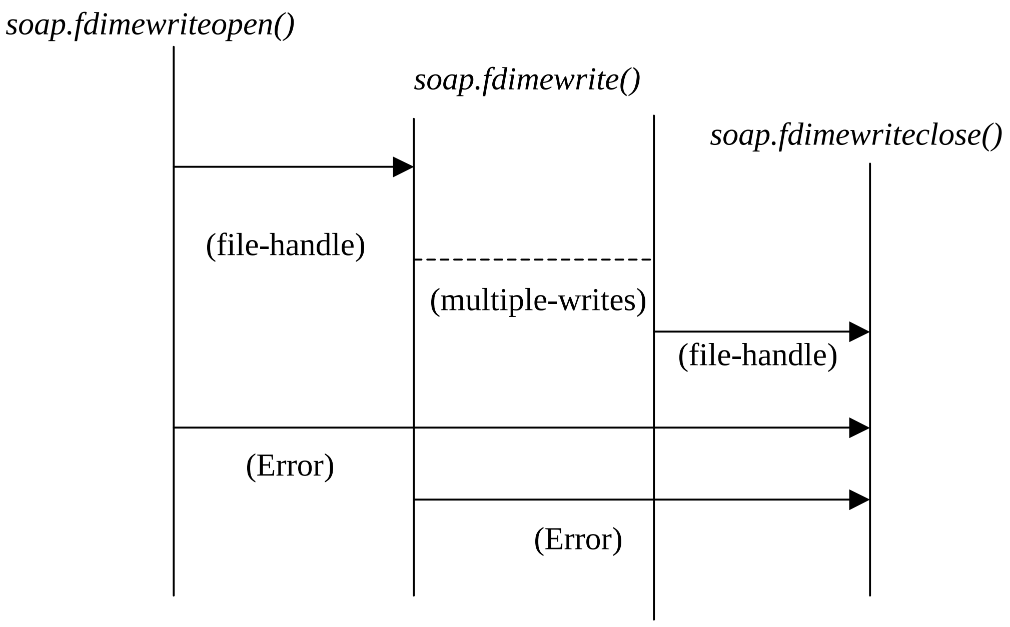


Fig: 9 – Dime Receive Streaming

DIME sending-side streaming:

The three function calls that are available are:

1. *soap.fdimereadopen( ):* Called by the gSOAP DIME attachment sender at run time to start reading from a data store for an outbound transmission. The contents are read in chunks at the run time with the multiple fdimeread() function calls and streamed into SOAP/XML/DIME output stream.

```
soap->fdimereadopen = dime_read_open;
static void *dime_read_open(struct soap *soap, void *handle, const
char *id, const char *type, const char *options)
```

The id field contains the image id, type field contains the image type i.e. “image/jpeg” and options field contains additional information about DIME attachment. In this case it contains the image name. The callback returns the handle that is next passed to soap.fdimeread() and soap.fdimereadclose() functions. In case of error the function returns NULL.

1. *soap.fdimeread():* Called by the gSOAP DIME attachment sender at run time to read data from the data store for streaming into the output stream. The prototype for this function is as follows:

   ```
   soap->fdimeread = dime_read;
   static size_t dime_read(struct soap *soap, void *handle, char *buf, size_t len)
   ```

   The handle field contains the file handle returned by the fdimereadopen() callback. The buffer of length len is used to store chunk of data. In case of successful operation the call back returns no of bytes read otherwise in case of any error, the call back returns 0. There could be multiple fdimeread() calls.

2. *soap.fdimereadclose():* Called by the gSOAP DIME attachment sender at run time to end the streaming process to close the data store. It is called either after successfully transmitting the data or after any error occurrence.

   ```
   soap->fdimereadclose = dime_read_close;
   static void dime_read_close(struct soap *soap, void *handle)
   ```

   The handle contains the file handle returned by the fdimereadopen() callback.

The following diagram explains the DIME send streaming –

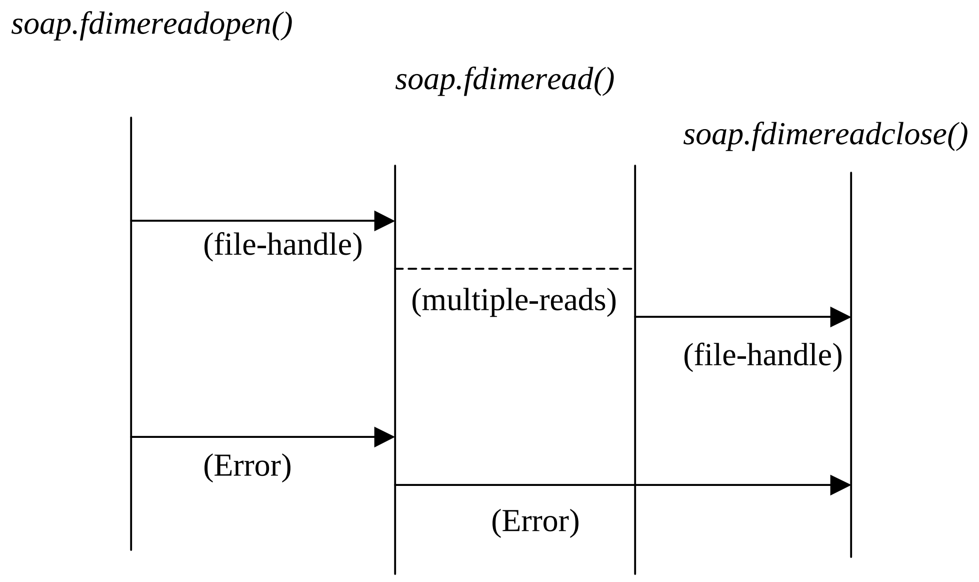


Fig: 10 – Dime Send Streaming

2.6 DIME Chunked Data Streaming

gSOAP automatically handles inbound chunked DIME attachments. gSOAP also supports the transmission of outbound-chunked DIME attachments without prior determination of DIME attachment size. To enable chunking the output-mode SOAP_IO_CHUNK flag must be set and __*size* field of an attachment must be set to zero. The DIME *fdimeread* callback then fetches data in chunks and it is important to fill the entire buffer unless the end of the data has been reached and the last chunk is to be send. That is, *fdimeread* should return the value of the last *len* parameter and fill the entire buffer *buf* for all chunks except the last.

3. Implementation Details

3.1 Use of gSOAP Stub and Skeleton Compiler to Build SOAP-DIME Client

Implementation of SOAP-DIME client requires a **stub** routine for each remote method that the client invokes. (Refer to figure-8). The gSOAP stub and skeleton compiler is a preprocessor, which generates the necessary C/C++ sources to build SOAP C++ clients. For this application the input to the gSOAP compiler is a header file. The name of the file is *gdx.h.* The SOAP service methods are specified in the *gdx.h* as function prototypes. **g**SOAP automatically generates the stub routines for these functions.

```
//Contents of the file:      gdx.h

//gsoap gdx service name: gdx gSOAP Data eXchange service
//gsoap gdx service namespace: http://www.genivia.com/services/gdx.wsdl
//gsoap gdx service location: http://www.cs.fsu.edu/~engelen/gdx.cgi
//gsoap gdx schema namespace: http://www.genivia.com/schemas/gdx.xsd


//USER authentication
struct gdx__Authentication
{ char *userid ;
  char *passwd ;
};
…..

int gdx__get(struct gdx__Authentication authentication, struct gdx__Resources
                   resources, struct gdx__Data &data );
int gdx__put(struct gdx__Authentication authentication, struct gdx__Resources
                  resources, struct gdx__Data data, struct gdx__putResponse{ } *  );
int gdx__del(struct gdx__Authentication authentication, struct gdx__Resources
                   resources, struct gdx__delResponse { } * );
int gdx__list(struct gdx__Authentication authentication, struct gdx__Resources
                   &resources );
```

The header file specifies the service details for the gSOAP compiler. The remote methods are declared as the gdx__get, gdx__put, gdx__del and gdx__list function prototype. The method specifies all the necessary details for the gSOAP compiler to generate the stub routine. The input parameters of the remote methods are objects of structures, which are passed by value. The last parameter of each remote method is the output parameter, which is passed by reference. On successful complementation, the method returns SOAP_OK otherwise an error code is returned.

Namespace Considerations:

All the remote methods use the namespace prefix *gdx*__. The purpose of the namespace prefix is to associate a remote method name with a service in order to prevent the naming conflicts i.e. to distinguish the remote methods with same name but used by the different services. The use of name space prefix is also required to enable SOAP applications to validate the contents of the SOAP messages. When the response arrives, the stub routine can verify the contents by using the information provided in the namespace-mapping table. The client application needs to include the namespace mapping table as a part of client application code. The stub accesses the table at run time while encoding and decoding the messages to resolve the namespace bindings.

```
//Contents of the file: gdx.nsmap

# include "soapH.h"
SOAP_NMAC struct Namespace namespaces[] =
{
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/",
                         "http://www.w3.org/*/soap-envelope"},
  {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/",
                         "http://www.w3.org/*/soap-encoding"},
  {"xsi", "http://www.w3.org/2001/XMLSchema-instance",
              "http://www.w3.org/*/XMLSchema-instance"},
  {"xsd", "http://www.w3.org/2001/XMLSchema", "http://www.w3.org/*/XMLSchema"},
  {"gdx", "http://www.genivia.com/schemas/gdx.xsd"},
   {NULL, NULL} //end of table.
};
```

The first four entries are standard entries used by SOAP. The later one is user specified. The last entry of the namespace prefix gdx__, which is bound to “http://www.genivia.com/schemas/gdx.xsd” by namespace mapping table, is used by stub to encode all the remote the methods. gSOAP does this task automatically by using the prefix gdx__ of all the remote methods, specified in the header file. Also the different structures that are declared in the file **‘gdx.h’** also have namespace prefix.

To generate the stub routines, the gSOAP compiler is invoked. The command used for this is as follows. Makefile has the list of all the commands.

```
soapcpp2 gdx.h
```

The stub routine can be called from the client program at any time to invoke the remote method. The stub routine is saved in the file *dimeclient.cpp.* In all there are four dimeclientX.cpp files, one for each remote method. The interface to the remote method gdx__get written in the stub is as follows.

```
// Contents of file dimeclient.cpp
….
….
int main(int argc, char ** argv)
{
      …….

if (soap_call_gdx__get(&soap, url, "",authentication,resources,data))
{
    soap_print_fault(&soap, stderr);
  }
 else
  {
     printf("\n Done ... gdx_get()  \n");
    ………
    }
     ……

     return SOAP_OK;

}
```

The parameters of the function soap_call_gdx__get are all identical with the parameters of gdx_get with three additional parameters. The first parameter is the pointer to gSOAP runtime environment. The second parameter is the URL, which is the endpoint URL, where the web service is located along with the valid port number. The third parameter is the SOAP action parameter. The fourth parameter is the object of the structure *gdx__Authentication*. This contains two fields where the first is username and the second is password. The next parameter is the object of structure *gdx__Resources*. This structure contains all the information about the files that are requested to either download, upload or delete by the client. The two fields of this structure are the file number along with the pointer to array of strings, which has all the file names to be processed. All the file names that the client wants to either upload or download or delete are stored in the linked list. Each node contains the filename along with some additional information. Before making the *soap__call* all the information from the linked list is stored in this structure. The last parameter is data, which is the output parameter. The structure contains the number of attachments send by the server long and the pointer to array of base64 structs. When successful the stub returns *SOAP_OK* and the output parameter contains the valid response otherwise the SOAP fault is displayed with *soap_print_fault* function. Similarly, the function soap_call_gdx__put, soap_call_gdx__list, soap_call_gdx__delete are created and handled.

Client proxy classes:

The proxy class is automatically generated by the gSOAP complier with the help of the information provided in header file *gdx.h.* The service name provided in the *gdx.h* file becomes the name of the proxy class. Here the name of the class is gdx. The default encoding is SOAP-RPC, but for this application the XSD type encoding is used which improves the interoperability.

```
//Contents of file : soapgdxProxy.h

#ifndef gdx_H
#define gdx_H
#include "soapH.h"
class gdx
{   public:
        struct soap *soap;
        const char *endpoint;
        gdx() { soap = soap_new(); endpoint =
"http://www.cs.fsu.edu/~engelen/gdx.cgi"; };
        ~gdx() { if (soap) { soap_destroy(soap); soap_end(soap); soap_done(soap);
free((void*)soap); } };

  int get(struct gdx__Authentication authentication, struct gdx__Resources
resources, struct gdx__Data &data) { return soap ? soap_call_
gdx__get(soap, endpoint, NULL, authentication, resources, data) : SOAP_EOM; };
…………

#endif
```

3.2 Use of gSOAP Stub and Skeleton Compiler to Build SOAP Web Services

The server implements four RPCs first is to download data from the web service (gdx__get), second is to upload data on the web service (gdx__put), third is to view the list of data files on the server (gdx__list) and fourth is to delete files on the server (gdx__delete). The specification of these four methods is as follows.

```
//Contents of file - gdx.h
int gdx__get(struct gdx__Authentication authentication, struct gdx__Resources
                  resources, struct gdx__Data &data );
int gdx__put(struct gdx__Authentication authentication, struct gdx__Resources
                 resources, struct gdx__Data data, struct gdx__putResponse{ } *  );
int gdx__del(struct gdx__Authentication authentication, struct gdx__Resources
                  resources, struct gdx__delResponse { } * );
int gdx__list(struct gdx__Authentication authentication, struct gdx__Resources
                  &resources );
```

To generate the skeleton routines, the gSOAP compiler is invoked. The command used for this is as follows.

```
soapcpp2 gdx.h
```

The compiler generates the skeleton routines for the get, put, del, and list remote methods as specified in the gdx.h header file. The skeleton routines are *soap_serve_gdx__get, soap_serve_gdx__put, soap_serve_gdx__del* and *soap_serve_gdx__list.* These are saved in the file *soapServer.cpp.* The soapC.cpp file generated contains serializers and deserializers for the skeleton. The *soap_serve* function written in the *soapServer.cpp* file handles client requests on the standard input stream and dispatches the remote method requests to the appropriate skeletons to serve the requests. The skeleton then calls the remote method implementation function. During runtime an extra parameter gets attached to each remote method. The new parameter is a pointer to gSOAP runtime environment.

WSDL Service:

The *soapcpp2* compiler generates a **W**eb **S**ervice **D**escription **L**anguage file for this service. The compiler produces the WSDL file for the set of remote methods. The name of the WSDL file is the namespace prefix that the remote methods have used. If there are multiple namespaces then the compiler will create multiple WSDL files. For this application the remote methods use *gdx* as namespace prefix. Hence the name of the WSDL file is *gdx.wsdl*. This file can be published with the help of which the client application can be built to use the web service.

The gSOAP compiler also generates the XML schema file for all the complex data types when declared with namespaces. For this application there are few structures that are declared with the namespace *gdx*. The *gdx.xsd* file contains the XML schema for these complex structures.

SOAP-DIME Web Service:

The web service first does some initialization work, creates a listening port and waits for the request from the client. There are various gSOAP functions that are used for this purpose.

Function calls	**Description**
*soap_init(**struct** soap *soap):*	Initializes the gSOAP Runtime environment
*soap_init1(**struct** soap *soap, int imode)*	Initializes the gSOAP runtime environment with and sets flags.
*soap_init2(**struct** soap *soap, int imode, int omode)*	Initializes the gSOAP runtime environment with and sets flags.
*soap_bind(**struct** soap*soap, **char** *host, **int** port, **int** backlog):*	This is the bind function. The value of parameter backlog is the maximum queue size for the requests. If the value oh parameter host = NULL then the host is the machine on which the web service is running. The return value is the active socket number.
*soap_accept(**struct** soap *soap):*	Returns the socket number after accepting the request.
*soap_end(**struct** soap *soap):*	Cleans up the decentralized and temporary data, which is created via soap_malloc function.
*soap_free(**struct** soap *soap):*	Cleans up the temporary data only.
*soap_done(**struct** soap *soap)*	Closes the sockets and removes the callbacks.

Multi-threaded web service:

It is necessary for the web service to be multithreaded to avoid the long response time. gSOAP supports the implementation of multi-threaded stand-alone services. The web service has a standard port available on which the web service can accept the client request. For each request the web service creates a thread, which is then assigned the work of serving the request. After the successful completion of the request the thread exits. The code does not wait for threads to join the main thread upon program termination.

The *soap_serve* method acts as the service dispatcher. It invokes the remote method via the skeleton routine to serve the SOAP client request.

```
//Contents of file : dimeserver.cpp

int main(int argc, char **argv)
{
  struct soap soap;
  soap_init(&soap);
  …….
  pthread_create(&tid, NULL, (void*(*)(void*))process_request, (void*)tsoap);
….
}

void *process_request(void *soap)
{
  pthread_detach(pthread_self());
  soap_serve((struct soap*)soap);
  ……
  return NULL;
}

int gdx__get(struct soap *soap,gdx__Authentication authentication, gdx__Resources
resources, gdx__Data &data)
{
 ………..
return SOAP_OK
```

The remote method gdx_get is explained below. With the help of this method the client can download the data from the web service.

gdx__get :

```
//Contents of file : dimeserver.cpp

int gdx__get(struct soap *soap, gdx__Authentication authentication, gdx__Resources
                  resources, gdx__Data &data)
{
    ……
   ……
  return SOAP_OK
}
```

The *gdx__get* method can send the data in three different ways. All the three approaches are discussed below.

Streaming the data WITH HTTP Chunking:

If the SOAP_IO_CHUNK flag is set and the *__size* field of an attachment is set to ZERO then gSOAP enables HTTP chunking. With this the data can be streamed without the need to know the data size. gSOAP automatically handles inbound chunked DIME attachments. If chunking is enabled then only data compression option is provided to improve the performance of the operation.

```
//Contents of file : dimeserver.cpp

if((soap->omode & SOAP_IO) == SOAP_IO_CHUNK)
{
      data.data[count].__ptr = (unsigned char*)fd;
      data.data[count].__size = 0;
      data.data[count].type = "";
      data.data[count].id = NULL;
      printf("\n Method#2 - Streaming the data WITH HTTP Chunking ....\n");

  }
```

Streaming the data WITHOUT HTTP Chunking:

If chunking is not enabled then the server web service streams the data without chunking. In that case the .*__ptr* field of the attachment is set to the file size that is to be transmitted. In both the cases the *type* field is set to “” to enable DIME.

```
//Contents of file : dimeserver.cpp

else if(!fstat(fileno(fd), &sb) && sb.st_size > 0)
    {
      data.data[count].__ptr =  (unsigned char*)fd;
      data.data[count].__size = sb.st_size;
      data.data[count].type = "" ;
      data.data[count].id = NULL;
      printf("\n Method#2 - Streaming the data WITHOUT HTTP Chunking ....\n");
      }
```

Sending the data as an attachment:

If chunking is not enabled or the file size is unknown in that case the file is buffered before sending and then send the file as an attachment. In case of the success the method returns SOAP_OK otherwise the method has to set *SOAP_FAULT*, which denotes an exception with details that can be defined by the user.

```
//Contents of file : dimeserver.cpp
else
    {
      printf("\n Method#3 - Can not stream so sending data as an attachment.\n");

      //MEMORY allocation for the image.
      image.__ptr = (unsigned char*)soap_malloc(soap, MAX_FILE_SIZE);
      data.data[count].__ptr = (unsigned char*)soap_malloc(soap, MAX_FILE_SIZE);
      c = (unsigned char *) malloc(sizeof(unsigned char));

      i=0;
      // COPY the file to the buffer.
      while(!feof(fd))
      {
        if(fread(c,sizeof(*c),1,fd))
        {
          image.__ptr[i] = *c;
          i++;
        }
            ……….
```

Similarly the other remote methods, gdx__put, gdx__list, gdx__delete are handled.

The list of all the files that gets created with the gSOAP stub and skeleton compiler is given below. The next sections discuss the file attachment handling and data streaming in details.

File Name	**Description**
soapH.h	Main header file included by both client and server.
soapC.cpp	Serializers and dserializers for specific data structures.
soapClient.cpp	Client stub routines and proxies for all the routines.
soapServer.cpp	Service skeleton routine.
soapStub.h	A modified header file created from the compiler input header file, gdx.h.
gdx.xsd	gdx.xsd file contains the XML schema for each namespace prefix gdx used by the data structure in the header file gdx.h.
gdx.wsdl	The file generated with the WSDL description for each namespace prefix gdx used by the remote method in the header file gdx.h.
gdx*.*.xml	Eight SOAP/XML request/response files are generated,
gdx.nsmap	The nsmap file generated for the namespace prefix gdx used by the remote method in the header file gdx.h, which is the input to the gSOAP compiler.

The serve provides two options for the data transmission when the client wants to download the data from the server with the remote method *gdx__get*. The two options are either send the data as an attachment or stream the data. Also the client can send the data while uploading the data on the server machine with the remote method *gdx__put* as an attachment or with the data streaming. This option is not provided for the remote methods *gdx__del* with which the client can delete the files on the server machine and *gdx__list* with which the client can get list of files from the server machine. Both these options are discussed in the section 3.3 and 3.4 below.

3.3 How the data streaming is done

This option is provided for the client and the server while data uploading or downloading. The advantages of data streaming are already discussed in the section-1. The DIME data streaming is done through function callbacks. The callbacks are described in the section 2.5. The following figure describes the DIME streaming for the RPC function gdx__get.

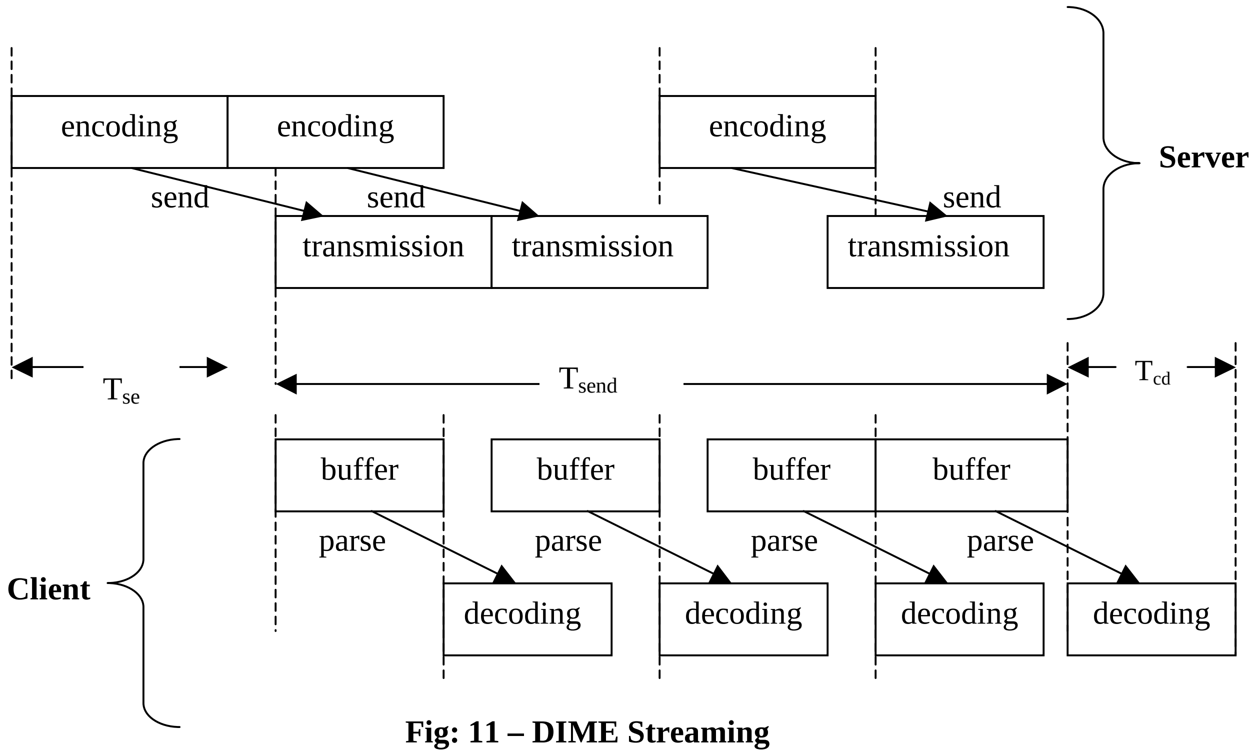


Fig: 11 – DIME Streaming

T_{se} - Time taken by server for encoding the SOAP message and collecting DIME data.

T_{send} - Total time for transmission and buffering of data.

T_{cd} – Time taken by client for decoding of SOAP messages and storing DIME data.

The figure-11 shows the streaming of data from server to client[8]. The encoding of the data to XML takes place by gSOAP compiler while the message is transferred in chunks of XML. At the server side each chunk is encoded and sent at the same time while the next data is being encoded. At the client side XML messages are parsed and then decoded as soon as the data arrives in the buffer. The size of the buffer is independent of the size of the chunk as the process of decoding starts immediately after the reception of data in buffer. The network bandwidth and the processor speed decided the chunk size. For this application the maximum chunk size is 32K. Similarly the gdx__put function handles the data streaming between the client and the server.

3.4 How the file attachment is handled?

This option is provided for the client to download the data from the server as a DIME attachment.

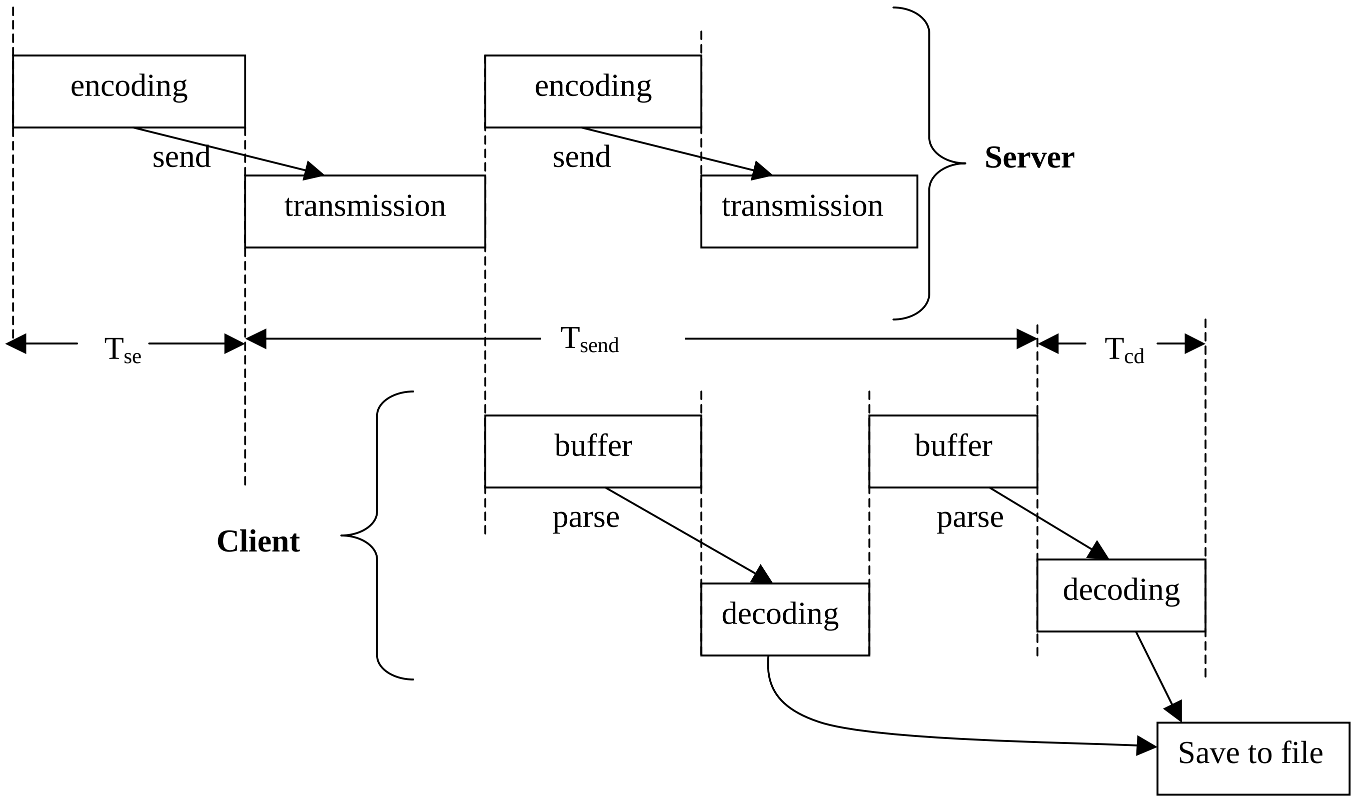


Fig: 12 – DIME File attachment handling

T_{se} - Time taken by server for encoding SOAP messages and collection DIME data.

T_{send} - Total time for transmission and buffering of data.

T_{cd} – Time taken by client for decoding of SOAP messages and storing DIME data.

The figure-12 shows the sending of data as an attachment from server to client[8]. The encoding of the data to XML takes place by gSOAP compiler then the message is transferred in chunks of XML. At the server side each chunk is encoded and then sent one at a time. The next data is then encoded and sent over. At the client side XML messages are parsed in the buffer and then the data is decoded. The size of the buffer is can be calculated from the size of the data from the DIME header. The network bandwidth and the processor speed decided the chunk size. For this application the maximum chunk size is 32K. Similarly the gdx__put function handles the data sent via file attachment between the client and the server.

3.5 System Usage

1. To build the SOAP web service:

The commands used to build the web service are –

```
>  ../soapcpp2 gdx.h
>  g++  -Wall -O2 -I../  -o dimeserver dimeserver.cpp soapC.cpp soapServer.cpp
../stdsoap2.cpp –lpthread -lz
```

2. To build SOAP client:

```
> ../soapcpp2 gdx.h
>  g++   -Wall -O2 -I../   -o dimeclient dimeclient.cpp soapC.cpp soapClient.cpp
   ../stdsoap2.cpp -lpthread -lz
>  g++   -Wall -O2 -I../  -o dimeclient1 dimeclient1.cpp soapC.cpp soapClient.cpp
   ../stdsoap2.cpp - lpthread -lz
>  g++   -Wall -O2 -I../   -o dimeclient2 dimeclient2.cpp soapC.cpp soapClient.cpp
   ../stdsoap2.cpp -  lpthread -lz
>  g++   -Wall -O2 -I../   -o dimeclient3 dimeclient3.cpp soapC.cpp soapClient.cpp
    ../stdsoap2.cpp -lpthread -lz
```

3. To execute the SOAP web service:

To enable streaming with chunking:

```
>  dimeserver  -SC 180012
```

To enable streaming with chunking with compression:

```
>  dimeserver  -SC  -C 180012
```

To enable streaming without chunking:

```
>  dimeserver  -ST 180012
```

To enable attachment:

```
> dimeserver -AT 180012
```

4. To execute the SOAP client:

To download files car.jpg home.jpg design.jpg from the server with streaming:

> dimeclient –ST car.jpg home.jpg design.jpg **http://linprog1.cs.fsu.edu:180012** newcar.jpg newhome.jpg newdesign.jpg

To download files car.jpg home.jpg design.jpg from the server as an attachment

> dimeclient –AT car.jpg home.jpg design.jpg **http://linprog1.cs.fsu.edu:180012** newcar.jpg newhome.jpg newdesign.jpg

To upload files tree.jpg, image.jpg, view.jpg on the server with HTTP chunking:

> dimeclient1 –SC tree.jpg image.jpg view.jpg **http://linprog1.cs.fsu.edu:180012**

To upload files tree.jpg, image.jpg, view.jpg on the server with HTTP chunking with compression:

> dimeclient1 –SC –C tree.jpg image.jpg view.jpg **http://linprog1.cs.fsu.edu:180012**

To upload files tree.jpg, image.jpg, view.jpg on the server with streaming:
> dimeclient1 –ST tree.jpg image.jpg view.jpg **http://linprog1.cs.fsu.edu:180012**

To upload files tree.jpg, image.jpg, view.jpg on the server with attachment:

> dimeclient1 –AT tree.jpg image.jpg view.jpg **http://linprog1.cs.fsu.edu:180012**

To get the list of files from the server :

> dimeclient2 –ST/-AT **http://linprog1.cs.fsu.edu:180012**

To delete files tree.jpg, image.jpg, view.jpg from the server :

> dimeclient3 -ST/-AT tree.jpg image.jpg view.jpg **http://linprog1.cs.fsu.edu:180012**

5. Future Work

1. File access permission

The current application allows any client to connect to the web service and the client has full access to the data on the web service. Therefore any client can view or delete all the files that are on the web service. Therefore there could be a situation where a malicious client is trying to delete the data from the web service. Therefore in future version the file access permission could be set for each user. The user who has uploaded the file only can delete the file. Also with the file access permissions only authorized users can be given the access to view it. Also in the future version clients can be given a directory where he/she can store their data. That particular client depending upon the contents of the directory could set the access permission for the directory. To set the permissions the web service will store the file information with the user credentials such as username and password with it. Before giving the full access to the directory the web service can check the username and password provided against the data stored in its database. If the user is valid user then the access is provided.

2. Secure Web-Service

If the client wants to store the data on the web service then it must provide the unique username and password to the web service for the future reference. While sending the user credentials proper encryption algorithm should be used to prevent the unwanted intrusions. There are various ways with which this functionality can be provided. The two different approaches are described below.

Secure Web-Service with HTTPS/SSL

The web service can be made secure by allowing it to support HTTPS/SSL when the web service is used as CGI. gSOAP supports OPENSSL. To enable OpenSSL, the code must be compiled with the option –DWITH_OPRNSSL. Call to the function *soap_ssl_accept* after *soap_accept* enables SSL support. In addition to this, the key file, DH file and password are needed. Function *soap_ssl_server* initializes the server side SSL. The *CRYPTO_thread_setup()* routine is used to setup locks for the multithreaded routine.

By default the server authentication is enabled. The gSOAP provides facility to unable the server authentication by setting appropriate flags. By default the client as not required to authenticate. But the application can support client authentication. The server is required to set specific flag in the call to above function.

HTTP Authentication:

Setting the soap.userid to username and soap.passwd strings to the password enables HTTP authentication at the client-side. A server may request user authentication and denies access (HTTP 401 error) when the client tries to connect without HTTP authentication (or with the wrong authentication information). A client MUST set the *soap.userid* and *soap.passwd* strings for each call that requires client authentication. The strings are reset after each successful or unsuccessful call.

A stand-alone gSOAP Web Service can enforce HTTP authentication upon clients, by checking the *soap.userid* and *soap.passwd* strings. These strings are set when a client request contains HTTP authentication headers. The strings SHOULD be checked in each service method.

```
//Contents of the file : dimeserver.cpp

int gdx__del(struct soap *soap, gdx__Resources resources, gdx__delResponse * resp)
{
  if (!soap->.userid || !soap->.passwd || strcmp(soap->.userid, "user#123") ||
      strcmp(soap->.passwd, "letmein"))
     return 401;
  ...
}
```

5. Conclusion

With the use of gSOAP an efficient distributed file access with web-service can be built. The application built on gSOAP can transmit the binary data with DIME attachments with or without streaming. With DIME output streaming, the binary data is retrieved from the data source at run time in parts without the need to store the entire file contents. SOAP uses HTTP as the transport mechanism and HTTP can pass through most of the firewalls therefore it is possible to do file exchange over such networks. SOAP uses XML message format and therefore SOAP has inherited all the advantages that XML provides. With the DIME attachments the date transfer between the systems that do not use XML format e.g. Electronic Data Interchange (EDI) is possible without converting the data to and from XML. With DIME there is no restriction on size and format of data.

6. References

[1] Aaron Skonnard. SOAP: The Simple Object Access Protocol, Microsoft Corporation, **http://www.microsoft.com/mind/0100/soap/soap.asp**

[2] Aaron Skonnard. Understanding SOAP, Microsoft Corporation, **http://msdn.microsoft.com/library/default.asp?url=/library/enus/dnsoap/html/understandsoap.asp**

[3] Gunjan Gupta, A Simple Object Access Protocol (SOAP) stub compiler for C, MS Project, 2000

[4] Jeannine Hall Gailey, Microsoft Corporation, Sending Files, Attachments, and SOAP Messages Via Direct Internet Message Encapsulation **http://msdn.microsoft.com/msdnmag/issues/02/12/DIME/**

[5] Jeannine Hall Gailey, Microsoft Corporation, Using Web Services Enhancements to Send SOAP Messages with Attachments **http://msdn.microsoft.com/library/default.asp?url=/library/enus/dnwse/html/wsedime.asp**

[6] Matt Powell. Microsoft Corporation, DIME: Sending Binary Data with Your SOAP Messages, **http://msdn.microsoft.com/library/default.asp?url=/library/en-** us/dnservice/html/service01152002.asp

[7] Robert van Engelen. gSOAP 2.6.0 User Guide http://www.cs.fsu.edu/~engelen/soapdoc2.html

[8] Robert van Engelen. Code Generation Techniques for developing light-weight XML web services for embedded devices, ACM SIGAPP SAC Conference, 2004

[9] Robert van Engelen. Pushing the SOAP Envelope with web services for scientific computing, 1st International Conference on Web-services, June 22-26, 2003 Las Vegas USA.

[10] W3C Note 11 December 2000, SOAP Messages with Attachments, **http://www.w3.org/TR/SOAP-attachments**

[11] W3C Working Draft 9 July 2001, SOAP Version 1.2, **http://www.w3.org/TR/2001/WD-soap12-20010709/#_Toc478383512**