gSOAP 2.2.3 User Guide

Robert van Engelen
Genivia inc. and Florida State University
engelen@acm.org

March 2, 2003

Contents

Includes HTTP, TCP/IP, XML, and DIME stacks.

Supports one-way messaging, including asynchronous send and receive operations.

Supports saving and loading of XML serialized C/C++ data structures to/from files.

The

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119.

3 Differences Between gSOAP Versions 2.1 (and Earlier) and 2.2

Run-time options and flags have been changed to enable separate recv/send settings for transport, content encodings, and mappings. The flags are divided into four classes: transport (IO), content encoding (ENC), XML marshalling (XML), and C/C++ data mapping (C). The old-style flags soap_disable_X and soap_enable_X, where X is a particular feature, are depricated. See Section 7.10 for more details.

4 Differences Between gSOAP Versions 1.X and 2.X

gSOAP versions 2.0 and higher have been rewritten based on versions 1.X. gSOAP 2.0 and higher

Function	Description
soap_init(**struct** soap *soap)	Initializes a runtime environment (required only once)
struct soap *soap_new()	

```
   soap_serve(&soap);
g
```

Or alternatively:

```
int
```

Endpoint URL: `http://services.xmethods.net:80/soap`
SOAP action: `""` (2 quotes)

Note that the parameters of the `soap_call_ns1__getQuote` function ns1 are idencat the _

```
soap_end(&soap); // clean up all deserialized data
...
```

This client composes an array of stock quotes by calling the `ns1__getQuote` stub routine for each symbol in a portfolio array.

This example demonstrated how easy it is to build a SOAP client with gSOAP once the details of a Web service are available in the form of a WSDL document.

method name specified in the getQuote.h header file. In general, if a function name of a remote method, **struct** name, **class** name, **enum** name, or field name of a **struct** or **class** has a pair of

The namespace prefix is separated from the name of a data type by a pair of underscores (__

```
#include "soapH.h"
class Quote
f public:
   struct soap *soap;
   const char *endpoint;
   Quote() f soap = soap_new(); endpoint = "http://services.xmethods.net/soap"; g;
   ~Quote() f if (soap) f soap_destroy(soap); soap_end(soap); soap_done(soap); free((void*)soap); gg;
   int getQuote(char *symbol, float &result) f return soap ? soap_call_ns__getQuote(soap, endpoint, "", symbol, result) : SOAP_EOM; g;
g;
```

The validation of this service response by the stub routine takes place by matching the namespace names (URIs) that are bound to the `xsd` namespace prefix. The stub also expects the `getQuoteResponse` element to be associated with URI `urn:xmethods-delayed-quotes` through the binding of the namespace prefix `ns1` in the namespace mapping table. getQuoteResponse337 -426(resp)-28(o337 us-quote0.

This use of a **struct** or **class**

```
<first>John</first>
<last>Doe</last>
</m:getNamesResponse>
...
```

where `first` and `last` are the output parameters of the getNames remote method of the service.

As another example, consider a remote method copy with an input parameter and an output parameter with identical parameter names (this is not prohibited by the SOAP 1.1 protocol). This can be specified as well using a response **struct**:

```
// Contente of file "copy.h":
int X_rox__copy_name(0 Td[(char*,)]TJ/F32 9.963 struct copy name, Tfr.568 0 Td[(*name,)]TJ/F32 9.963 Tt040
```

parameters. The remote method name is `getFlightInfo` and the method has two string parameters: the airline code and flight number, both of which must be encoded as `xsd:string` types. The method returns a `getFlightResponse` response element with a `return` output parameter that is of complex type

```
Content-Length:  634
SOAPAction:  "urn:galdemo:flighttracker"

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns:ns1="urn:galdemo:flighttracker"
  xmlns:ns2="http://galdemo.flighttracker.com"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
```

```
cout << r.return_.equipment << " flight " << r.return_.airline << r.return_.flightNumber
    << " traveling " << r.return_.speed << " mph " << " at " << r.return_.altitude
    << " ft, is located " << r.return_.currentLocation << endl;
```

This code displays the service response as:

```
A320 flight UAL184 traveling 497 mph at 37000 ft, is located 188 mi W of Lincoln,
NE
```

6.1.14 How to Specify a Method with No Input Parameters

To specify a remote method that has no input parameters, just provide a function prototype with one parameter which is the output parameter. However, some C/C++ compilers (notably Visual C++^{TM}) will not compile and complain about an empty **struct**. This **struct** is generated by gSOAP to contain the SOAP request message. To fix this, provide one input parameter of type **void*** (gSOAP can not serialize void* data). For example:

```
struct ns3
```

6.2 How to Use the gSOAP Stub and Skeleton Compiler to Build SOAP Web Services

```
// Contents of file "calc.cpp":
#include "soapH.h"
#include <math.h> // for sqrt()
main()
{
   soap_serve(soap_new()); // use the remote method request dispatcher
}
// Implementation of the "add" remote method:
int ns__add(struct soap *soap, double a, double b, double &result)
{
   result = a + b;
   return
```

This service application can be readily installed as a CGI application. The service description would be:

Endpoint URL:	the URL of the CGI application
SOAP action:	`""` (2 quotes)
Remote method namespace:	`urn:simple-calc`
Remote method name:	`add`
Input parameters:	`a` of type `xsd:double` and `b` of type `xsd:double`
Output parameter:	`result` of type `xsd:double`
Remote method name:	`sub`
Input parameters:	`a` of type `xsd:double` and `b` of type `xsd:double`
Output parameter:	`result` of type `xsd:double`
Remote method name:	`sqrt`
Input parameter:	`a` of type `xsd:double`
Output parameter:	`result` of type `xsd:double` or a SOAP Fault

The soapcpp2

```
          soap_end(soap_thr[i]); // deallocate data of old thread
        g
        soap_thr[i]->socket = s;
        pthread_create(&tid[i], NULL, (void*(*)(void*))soap_serve, (void*)soap_thr[i]);
      g
    g
  g
  return 0;
g
```

g
void

WSDL file. If multiple namespace prefixes are used to define remote methods, multiple WSDL files will be created and each file describes the set of remote methods belonging to a namespace prefix.

In addition to the generation of the `ns.wsdl`

```
  xmlns:tns="http://location/Service.wsdl"
  xmlns:ns="http://tempuri.org">
<types>
  <schema
    xmlns="http://www.w3.org/2000/10/XMLSchema"
    targetNamespace="http://tempuri.org"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
    <complexType name="addResponse">
      <all>
        <element name="result" type="double" minOccurs="0" maxOccurs="13/hr/>
         name="addResponse">
          <all>
            <element name="result" type="double" minOccurs="0" maxOccurs="13/hr/>
             name="addResponse">
              <element name="result" type="double" minOccurs="0" maxOccurs="13/hr/>
  <operation name="sub">
    <input message="tns:subRequest"/>
    <output message="tns:subResponse"/>
  </operation>
  <operation name="sqrt">
    <input message="tns:sqrtRequest"/>
    <output message="tns:sqrtResponse"/>
  </operation>
</portType>
<binding name="ServiceBinding" type="tns:ServicePortType">
  <SOAP:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="add">
    <SOAP:operation soapAction="http://tempuri.org#add"/>
    <input>
      <SOAP:body use="encoded" namespace="http://tempuri.org"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <SOAP:body use="encoded" namespace="http://tempuri.org"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
  <operation name="sub">
    <SOAP:operation soapAction="http://tempuri.org#sub"/>
    <input>
      <SOAP:body use="encoded" namespace="http://tempuri.org"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <SOAP:body use="encoded" namespace="http://tempuri.org"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
  <operation name="sqrt">
    <SOAP:operation soapAction="http://tempuri.org#sqrt"/>
    <input>
      <SOAP:body use="encoded" namespace="http://tempuri.org"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <SOAP:body use="encoded" namespace="http://tempuri.org"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
</binding>
<service name="Service">
  <port name="ServicePort" binding="tns:ServiceBinding">
    <SOAP:address location="http://location/Service.cgi"/>
  </port>
</service>
</definitions>
```

6.2.7 How to Import WSDL Service Descriptions

Note: see README.txt in the wsdlcpp directory for installation instructions for the importer.

The creation of SOAP Web Service clients from a WSDL service description is a two-step process.

First, execute the java wsdlcpp [illegible] soap

```
quotex.cgi AOL uk
```

returns the quote of AOL in uk pounds by communicating the request and response quote from the CGI application. See `http://xmethods.com/detail.html?id=5` for details on the currency abbreviations.

When combining clients and service functionalities, it is required to use one header file input to the compiler. As a consequence, however, stubs and skeletons are available for all remote methods, while the client part will only use the stubs and the service part will use the skeletons. Thus, dummy implementations of the unused remote methods need to be given which are never called.

Three WSDL files are created by gSOAP: `ns1.wsdl`, `ns2.wsdl`, and `ns3.wsdl`. Only the `ns3.wsdl` file

and if the data contains cycles. The second function (`soap_put`) generates the SOAP encoding output for that data type.

The function names are specific to a data type. For example, `soap_serialize_float(&soap, &d)` is called to serialize an **float** value and `soap_put`

This produces:

```
<ns:element-name xmlns:SOAP-ENV="..." xmlns:SOAP-ENC="..." xmlns:ns="..."
  ...  xsi:type="ns:type-name">
<name xsi:type="xsd:string">...</name>
</ns:element-name>
```

The serializer is initialized with the soap_

In principle, encoding MAY take place without calling the soap_serialize functions. However, as the following example demonstrates the resulting encoding is not SOAP 1.1 compliant. However, the messages can still be used with gSOAP to save and restore data in XML.

Consider the following **struct**:

```
// Contents of file "tricky.h":
struct Tricky
f
  int *p;
  int n;
  int
```

```
T1 var1;
T2 var2;
struct soap soap;
...
soap_init(&soap); // initialize at least once
[soap_imode(&soap, flags);] // set input-mode flags
soap_begin(&soap); // begin new decoding phase
[soap.recvfd = an_input_stream;]
soap_begin_
```

```
   public:
   xsd__string street;
   xsd__unsignedInt number;
   xsd__string city;
g;
class ns__Person
f
   public:
   xsd__Name name;
   enum ns
```

```
  soap_end(&soap);
  soap_done(&soap);
g
struct Namespace namespaces[] =
f
  f"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"g,
</address>
</father>
</johnnie>
```

The following program fragment decodes this content from standard input and reconstructs the orignal data structure on the heap:

```
#include "soapH.h"
int main()
f
   struct soap soap;
   ns_
   ns__Person *mother, *father, john;
   soap_init(&soap);
   soap_imode(&soap, SOAP_ENC_ZLIB); // optional
   soap_begin(&soap);
   soap_begin_recv(&soap);
   soap_default_ns__Person(&soap, &john);
   if (soap_get_ns__Person(&soap, &john, "johnnie", NULL))
      ... error ...
   ...
g
```

struct1 0 0 1 32.118 0 cn.(-332(NIyamespaceer,)-3namespaces[]er,)-3=nal

Caution: SOAP 1.2 requires the use of SOAP_ENV__Code, SOAP_ENV__Reason, and SOAP_ENV__Detail fields in a SOAP_ENV__Fault fault struct, while SOAP 1.1 uses faultcode, faultstring, and detail fields. Use soap_receiver_fault(**struct** soap *soap, **const char** *faultstring, **const char** *detail)

```
...
// A remote method invocation:
   soap_call_some_remote_
```

```
g;
...
```

When the gSOAP service is compiled and installed as a CGI application, the soap

C and C++ programming statements All class methods of a class should be declared within the class declaration in the header file, but the methods should not be implemented in code. All class method implementations must be defined within another C++ source file and linked to the application.

In addition, the following data types cannot be used in the header file (they can, however be used as a class method return type and as class method parameter types of a class declared in the header file):

union

Caution: The SOAP_XML_TREE

```
typedef int xsd__int;
class X f ... g;
class ArrayOfint f xsd__int *__ptr; int __size; g;
ns__
```

```
      char *msg = (char*)soap_malloc(soap, 1024); // allocate temporary space for detailed message
      sprintf(msg, "...", ...); // produce the detailed message
      return soap_receiver_fault(soap, "An exception occurred", msg); // return the server-side fault
   g
   ...
gsoap
```

```
struct soap *soap; // set by soap_new_ns__myClass()
char *name;
void setName(const char *s);
...
g         const ch[illegible]
```

Caution: The client and server applications may run slow due to the logging activity.

Caution: When installing a CGI application on the Web with debugging activated, the log files may

8 The gSOAP Remote Method Specification Format

A SOAP remote method is specified as a C/C++ function prototype in a header file. The function is REQUIRED to return **int**, which is used to represent a SOAP error code, see Section 8.2. Multiple remote methods MAY be declared together in one header file.

outparam is the declaration of an output parameter of the remote method

The general form of a remote method specification with a response element declaration for (multiple) output parameters is:

> [**int**] [namespace_prefix__]method_name([inparam1, inparam2, ...,] **struct** [namespace_prefix__]response_element_name
> *f*outparam1

8.3 C/C++ Identifier Name to XML Name Translations

One of the \secrets" behind the power and flexibility of gSOAP's encoding and decoding of remote method names, class names, type identifiers, and struct or class fields is the ability to specify namespace prefixes with these names that are used to denote their encoding style. More specifically, a C/C++ identifier name of the form

[namespace_

8.4 Namespace Mapping Table

A namespace mapping table MUST be defined by clients and service applications. The mapping table is used by the serializers and deserializers of the stub and skeleton routines to produce a valid

```
struct Namespace namespacesTable1[] = f
```

enables the implementation of built-in XML schema types (also known as XSD types) such as `positiveInteger`, `xsd:anyURI`, and `xsd:date`

```
typedef bool xsd__boolean;
```

Type xsd__boolean declares a Boolean (0 or 1), which is encoded as

```
<xsd:boolean xsi:type="xsd:boolean">...</xsd:boolean>
```

`xsd:byte` Represents a byte (-128...127). The corresponding type declaration is:

```
typedef char xsd__byte;
```

Type xsd__byte declares a byte which is encoded as

```
<xsd:byte xsi:type="xsd:byte">...</xsd:byte>
```

`xsd:dateTime` Represents a date and time. The lexical representation is according to the ISO 8601 extended format CCYY-MM-DDThh:mm:ss where "CC" represents the century, "YY" the year, "MM" the month and "DD" the day, preceded by an optional leading "-" sign to

```
<xsd:double xsi:type="xsd:double">...</xsd:double>
```

`xsd:duration` Represents a duration of time. The lexical representation for duration is the ISO 8601 extended format PnYn MnDTnH nMnS, where nY represents the number of years, nM the number of months, nD the number of days, T is the date/time separator, nH the number of hours, nM the umbe

Another possibility is to use strings to represent unbounded integers and do the translation in code.

`xsd:long` Corresponds to a 64-bit integer in the range -9223372036854775808 to 9223372036854775807. The type declaration is:

typedef long long xsd__long;

Or in Visual C++:

typedef LONG64 xsd__long;

Type xsd__long declares a 64-bit integer which is encoded as

```
<xsd:long xsi:type="xsd:long">...</xsd:long>
```

`xsd:negativeInteger` Corresponds to a negative unbounded integer (< 0). Since C++ does not support unbounded integers as a standard feature, the recommended type declaration is:

typedef long long xsd__negativeInteger;

Type

```
typedef char *xsd__normalizedString;
```

Type `xsd__normalizedString` declares a string type which is encoded as

```
<xsd:normalizedString xsi:type="xsd:normalizedString">...</xsd:normalizedString>
```

It is solely the responsibility of the application to make sure the strings do not contain carriage return (#xD), line feed (#xA) and tab (#x9) characters.

`xsd:positiveInteger` Corresponds to a positive unbounded integer (0). Since C++ does not support unbounded integers as a standard feature, the recommended type declartupegers-332(feature,2 9

xsd:positiveInteger

`xsd:token` Represents tokenized strings. Tokens are strings that do not contain the line feed (#xA) nor tab (#x9) characters, that have no leading or trailing spaces (#x20) and that have no internal sequences of two or more spaces. It is recommended to use strings to store

```
<xsd:unsignedShort xsi:type="xsd:unsignedShort">...</xsd:unsignedShort>
```

Other XML schema types such as `gYearMonth`, `gYear`, `gMonthDay`, `gDay`, `xsd:gMonth`, `QName`, `NOTATION`, etc., can be encoded similarly using a **typedef** declaration.

9.2.1 How to Use Multiple C/C++ Types for a Single Primitive XSD Type

Trailing underscores (see Section 8.3) can be used in the type name in a **typedef** to enable the

class xsd__anyURI_

Note the use of the trailing underscores for the **class** names to distinhuish the **typedef** type names from the **class** names. Only the most frequently used built-in schema types are shown. It is also allowed to include the xsd:base64Binray and xsd:hexBinary types in the hierarchy:

```
class xsd__base64Binary: public xsd__anySimpleType { public: unsigned char *__ptr; int __size; };
class xsd__hexBinary: public xsd__anySimpleType { public: unsigned char *__ptr; int __size; };
```

See Sections 9.9 and 9.10.

Methods are allowed to be added to the classes above, such as constructors and getter/setter methods.

Wrapper structs are supported as well, similar to wrapper classes. But thay cannot be used to

not indicate the possible loss of precision of floating point values due to the textual representation

The proxy of the remote method is used by a client to request a piece of information and the service responds with:

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

<SOAP-ENV:Envelope Tf 27.91 e:SOAP-ENV="http://schemas..91soap.org/soap/envelope/"
  .91 e:SOAP-ENC="http://schemas..91soap.org/soap/encoding/"
  .91 e:xsi="http://www.w3.org/1999/XMLSchema-instance"
  .91 e:xsd="http://www.w3.org/1999/XMLSchema"
<SOAP-ENV:Body>
<getInfoResponse>
<detail>
```

9.2.7 INF, -INF, and NaN Values of float and double Types

The gSOAP runtime stdsoap2.cpp and header file stdsoap2.h

9.3.4 How to "Reuse" Symbolic Enumeration Constants

A well-known deficiency of C and C++ enumeration types is the lack of support for the reuse of symbolic names by multiple enumerations. That is, the names of all the symbolic constants

enum SOAP_ENC__boolean *fg*;

The value `0`, for example, is encoded with an integer literal:

```
<SOAP-ENC:boolean xsi:type="SOAP-ENC:boolean">0<SOAP-ENC:boolean>
```

9.3.6 Bitmask Enumeration Encoding and Decoding

T4n>

Certain fields of a **class** can be (de)serialized as XML attributes. See 9.5.5 for more details.

A **class** instance is encoded as:

```
<[namespace-prefix:]class-name xsi:type="[namespace-prefix:]class-name">
<basefield-name1 xsi:type="...">...</basefield-name1>
<basefield-name2 xsi:type="...">...</basefield-name2>
...
<field-name1 xsi:type="...">...</field-name1>
<field-name2 xsi:type="...">...</field-name2>
...
</[namespace-prefix:]class-name>
```

where the `field-name` accessors have element-name representations of the class fields and the `basefield-name`

The namespace URI of the namespace prefix ns must be defined by a namespace mapping table, see Section 8.4.

9.5.2 Initialized static const Fields

A data member field of a class declared as **static const** is initialized with a constant value at compile time. This field is encoded in the serialization process, but is not decoded in the deserialization process. For example:

```
// Contents of file "triangle.h":
class ns
```

The following example declares Base and Derived classes and a remote method that takes a pointer to a Base class instance and returns a Base class instance:

```
// Contents of file "derived.h"
class Base
f
   public:
   char *name;
   Base();
   virtual void print();
g;
class Derived : public Base
f
   public:
   int num;
   Derived();
   virtual void print();
g;
int method(Base *in, struct methodResponse f Base *out; g &result);
```

This header file specification is processed by the gSOAP compiler to produce the stub and skeleton

// Content3us 7"der

pmethp *in,

p
clas"der

pmetho11d(Base *in,

p
clas]TJ/F32der

prinder;   pmetho*in,

p
clas]TJ/F32der   p134

prinder   p

pumder

```
int main()
f
   struct soap soap;
   soap_init(&soap);
   Derived obj1;
   Base :at*(f)]TJ/F32 9.963 TJ 0 -11.655 Td[(struct)]TJ/F31 9.963 Tf 29.744 0 TdmethodResponBaserap;
   soap
```

This header file specification is processed by the gSOAP stub and skeleton compiler to produce skeleton routine which is used to implement a service (so the client will still use the derived classes).

The method implementation of the Base class are:

then `x:def` is converted to "URI":def where `"URI"` is the namespace URI bound to `x` in the message received.

Because a remote method request and response is essentially a struct, XML attributes can also be associated with method requests and responses. For example

```
typedef char *xsd__string;
int ns__myMethod(@ xsd__string ns__name, ...);
```

```
    xsd__string value;
    struct ns__list *next;
g;
```

9.8 Dynamic Arrays

As the name suggests, dynamic arrays are much more flexible than fixed-size arrays and dynamic arrays are better adaptabe to the SOAP encoding and decoding rules for arrays. In addition, a typical C application allocates a dynamic array using malloc, assigns the location to a pointer variable, and deallocates the array later with free. A typical C++ application allocates a dynamic array using new, assigns the location to a pointer variable, and deallocates the array later with delete. Such dynamic allocations are flexible, but pose a problem for the serialization of data: how

The deserializer of a dynamic array can decode partially transmitted and/or SOAP sparse arrays,

```
g
ServiceArray::~ServiceArray()
f
   if (_ _ptr)
      free(_ _ptr);
   _
```

```
    __size = 0;
    __offset = 1;
g
Vector::Vector(int n)
```

```
// Contents of file "matrix.h":
class Matrix
{
   public:
   Vector *__ptr;
   int __size;
   int __offset;
   Matrix();
   Matrix(int n, int m);
   ~Matrix();
   Vector& operator[](int i);
};
```

For example, the following declaration specifies a matrix class:

```
typedef double xsd__double;
class Matrix
{
   public:
   xsd__double *__ptr
```

end of the list is reached, the buffered elements are copied to a newly allocated space on the heap for the dynamic array.

A list (de)serialization is also in affect for dynamic arrays when the pointer field does not refer to

Type *__ptrarray_elt_name

```
unsigned char *_ _ptr;
_int _
```

The following example in C/C++ reads from a raw image file and encodes the image in SOAP using the `base64Binary` type:

```
...
FILE *fd = fopen("image.jpg", "r");
xsd__base64Binary image(filesize(fd));
fread(image.location(), image.size(), 1, fd);
fclose(fd);
soap_begin(&soap);
image.soap_serialize(&soap);
image.soap_put(&soap, "jpegimage", NULL);
soap_end(&soap);
...
```

where filesize is a function that returns the size of a file given a file descriptor.

Reading the `xsd:base64Binary` encoded image.

```
...
xsd__base64Binary image;
soap_begin(&soap);
image.get(&soap, "jpegimage");
soap_end(&soap);
...
```

(base64) should be used for `SOAP-ENC:base64` schema type instead

d _ class.55 Tdd[(a1n9 2.989 0.398 re f -51.413 d[(a1n9 2.989 0.398 re f 1 0 0 1 -12.453 d[(a1n97 0 37-ss371.236 -13.549Tf 19XML3h4Tf 87.1

The **struct** or **class** name soap

9.11 Doc/Literal XML Encoding Style

gSOAP supports doc/literal SOAP encoding of request and/or response messages. However, the XML schema of the message data must be known in order for the gSOAP compiler to generate the

```
  f"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"g,
  f"xsi", "http://www.w3.org/2001/XMLSchema-instance", "http://www.w3.org/*/XMLSchema-instance"g,
  f"xsd", "http://www.w3.org/2001/XMLSchema", "http://www.w3.org/*/XMLSchema"g,
  fNULL, NULLg
g;
```

The SOAP request is:

```
   else
      printf("Time = %s\n", t);
   return 0;
g
```

9.11.1 Serializing and Deserializing XML Into Strings

To declare a literal XML \type" to hold XML documents in regular strings, use:

```
typedef char *XML;
```

To declare a literal XML \type" to hold XML documents in wide character strings, use:

```
typedef wchar_t *XML;
```

Note: only one of the two storage formats can be used. The differences between the use of regular strings versus wide character strings for XML documents are:

- Regular strings for XML documents MUST hold UTF-8 encoded XML documents. That is, the string MUST contain the proper UTF-8 encoding to exchange the XML document in SOAP messages.
- Wide character strings for XML documents SHOULD NOT hold UTF-8 encoded XML doc-

```
          <XMLDoc xmlns="http://my.org/mydoc.xsd">
            ...
          </XMLDoc>
        </ns:Document>
      </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
```

Important: the literal XML encoding style MUST be specified by setting `soap.encodingStyle`, where `soap` is a variable that contains the current runtime environment. For example, to specify no constraints on the encoding style (which is typical) use NULL:

```
struct SOAP_ENV__Fault
{
   char *faultcode; // MUST be string
   char *faultstring; // MUST be string
   char *faultactor;
   Detail *detail; // new detail field
   char *SOAP_ENV__Code; // MUST be string
   char *SOAP_ENV__Reason; // MUST be string
   char *SOAP_ENV__Detail; // MUST be string
   Detail SOAP_ENVB Detail; // new SOAP 1.2 detail field
};
```

where `Detail` is some data type that holds application specific data such as a stack dump.

When the skeleton of a remote method returns an error (see Section 8.2), then

11 SOAP Header Processing

A predeclared standard SOAP Header data structure is generated by the gSOAP stub and skeleton compiler for exchanging SOAP messages with SOAP Headers. This predeclared data structure is:

```
struct SOAP_ENV__Header
{ void *dummy;
};
```

which declares and empty header (some C and C++ compilers don't accept empty structs so a transient dummy field is provided).

client side, the `soap.actor` attribute can be set to indicate the recipient of the header (the SOAP `SOAP-ENV:actor` attribute).

A Web service can read and set the SOAP Header as follows:

```
int main()
f
  struct soap soap;
  soap.actor = NULL; // use this to accept all headers (default)
  soap.actor = "http://some/actor"; // accept headers destined for "http://some/actor" only
  soap_serve(&soap);
g
...
int method(struct soap *soap, ...)
f
  if (soap->header) // a Header was received
    ... = soap->header->t__
```

12 DIME Attac essing

gSOAP can transmit bina E attachments with or without streaming. With DIME output streaming, the bin ved from an application's data source at run time in parts without storing the ith DIME input streaming, the binary data will be handed to the application i eaming is implemented with function callbacks. Se Section 12.2 for more details

12.1 Non-Streaming DI

Without streaming, the binary da nted `xsd:base64Binary` and `xsd` structs/classes. These structs/classe fields: an id field for erencing (typically a content id (CID) binary data, and an options field to piggy DIME attachment support is fully automatic e of attachments at run time and use SOAP in DIM

O7(data)-7elds:ebeof theith l-44592 ,5432(functioy)]TJ/F32

A `xsd:base64Binary` type with DIME attachment support is decla y

77nsWhen4455laretyps5nn/F3232 9.963 Tf -71.86669.02cifyon

struct xsd__base64Binary

type f87. 321 0 Td[(t)28(22(0 -18unsign34(declacses37 rd[(struct)]TJ/F31 65.14e

```
   soap.fdimewrite = dime_write;
   soap_call_ns_ _method(&soap, ...);
   ...
g
void *dime_write_open(struct soap *soap, const char
                                            gconstr
```

and namespace mapping table files do not need to be modified by hand (Sections 6.2.5 and 8.4).

```
soapcpp2 quotex.h
```

the WSDL of the new `quotex` Web Service is saved as `quotex.wsdl`. Since the service name (`quotex`), location (`http://www.cs.fsu.edu/~engelen`

In this example, **class** ns__myClass has three transient fields: b, s, and n which will not be (de)serialized in SOAP. Field n

```
if (*soap->type && soap_match_tag(soap, soap->type, type))
f
  soap
```

The following example uses I/O function callbacks for customized serialization of data into a buffer and deserialization back into a datastructure:

```
char buf[10000]; // XML buffer
int len1 = 0; // #chars written
int len2 = 0; // #chars read
// mysend: put XML in buf[]
int mysend(structar
```

g

The `soap_done` function can be called to reset the callback to the default internal gSOAP I/O and HTTP handlers.

The following example illustrates customized I/O and (HTTP) header handling. The SOAP request is saved to a file. The client proxy then reads the file contents as the service response. To perform this trick, the service response has exactly the same structure as the request. This is declared by the **struct** `ns__test` output parameter part of the remote method declaration. This struct resembles the service request (see the generated `soapStub.h` file created from the header file).

The header file is:

```
//gsoap ns service name: callback
//gsoap ns service namespace: urn:callback
struct ns__
```

```
skip custom header
      return SOAP_EOF;
   return SOAP_OK;
g
main()
f
   structsoap soap;
   struct ns__test r;
   struct ns__person p;
   soap_init(&soap); // reset
   p.name = "John Doe";
   p.age = 99;
   soap.fopen = myopen; // use custom open
   soap.fpost = mypost; // use custom post
   soap.fparse = myparse; // use custom response parser
   soap.fclose = myclose; // use custom close
   soap_call_ns        ns
```

```
soap.fignore = myignore;
soap_call_ns__method(&soap, ...); // or soap_serve(&soap)
...
struct Namespace namespaces[] =
f
   f"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"g,
```

Increase the buffer size SOAP_BUFLEN by changing the SOAP_BUFLEN macro in stdsoap2.h. Use buffer size 65536 for example.

Use HTTP keep-alive at the client-side, see 13.10, when the client needs to make a series

This setting will not generate a sigpipe but read/write operations return SOAP_EOF instead. Note that Win32 systems do not support signals and lack the MSG_NOSIGNAL flag. The sigpipe handling and flags are not very portable.

A connection will be kept open only if the request contains an HTTP 1.0 header with "`Connection: Keep-Alive`" or an HTTP 1.1 header that does not contain "`Connection: close`" [illegible] a gSOAP client [illegible] the to 1thod call should use

sed.

```
((struct soap*)soap)->send_timeout = 60; // Timeout after 1 minute stall on send
soap_serve((struct soap*)soap);
soap_destroy((struct soap*)soap);
soap_end((struct soap*)soap);
soap_free((struct soap*)soap);
free(soap);
returnre f   1 0 0 1 0 -11p
```

13.17 Secure SOAP Clients with HTTPS/SSL

You need to install the OpenSSL library on your platform to enable secure SOAP clients to utilize HTTPS/SSL. After installation, compile all the sources of your application with option -DWITH_OPENSSL. For example on Linux:

```
g++ -DWITH_OPENSSL myclient.cpp stdsoap.cpp soapC.cpp soapClient.cpp -lssl -lcrypto
```

or Unix:

```
g++ -DWITH_OPENSSL myclient.cpp stdsoap.cpp soapC.cpp soapClient.cpp -lxnet -lsocket -lnsl
-lssl -lcrypto
```

or you can add the following line to soapdefs.h:

```
#define WITH_OPENSSL
```

and compile with option -DWITH_SOAPDEFS_H to include soapdefs.h

```
g++ -DWITH_OPENSSL -o myprog myprog.cpp stdsoap2.cpp soapC.cpp soapServer.cpp -lssl -lcrypto
```

Let's take a look at an example SSL secure multi-threaded stand-alone SOAP Web Service:

```
int main()
f
  int m, s;
  pthread_t tid;
  struct soap soap, *tsoap;
  soap_init(&soap);
  // soap.rsa = 1; // use RSA (or use DH which requires a DH file: see below)
  soap.keyfile = "server.pem"; // must be resident key file
  soap.cafile = "cacert.pem"; // must be resident CA file
  soap.dhfile = "dh512.pem"; // if soap.rsa == 0, use DH with resident DH file
  soap.password = "password"; // password
  soap.randfile = "random.rnd"; // (optional) some file with random data to seed PRNG
  m = soap
```

In case Web services have to verify clients, use a keyfile, CA file, a file with random data, and password in an SSL-enabled client:

```
...
soap_init(&soap);
soap.keyfile = "client.pem";
soap.password = "password";
soap.cafile = "cacert.pem";
soap.randfile = "random.rnd";
if (soap_call_ns__method(&soap, "https://linprog2.cs.fsu.edu:18000", "", ...)
...
```

Make sure you have signal handlers set in your service and/or client applications to catch broken connections (SIGPIPE):

```
signal(SIGPIPE, sigpipe
```

Answer the rest of the questions intelligently. The common name would be how this certificate might be referred o9331(F)83(orred)-2example, gSOAP supports two compression formats: deflate and gzip. The gzip format is used by default. The gzip format has several benefits over deflate. Firstly, gSOAP can automatically detect gzip compressed inbound messages, even without HTTP headers, by checking for the presence of a gzip header in the message content. Secondly, gzip includes a CRC32 checksum to ensure messages have been correctly received. Thirdly, gzip compressed content can be decompressed with other compression software, so you can decompress XML data saved by gSOAP in gzip format.

Gzip compression is enabled by compiling the sources with -DWITH_GZIP. To transmit gzip compressed SOAP/XML data, set the output mode flags to SOAP_ENC_ZLIB. For example:

```
soap
```

To restrict the compression to the deflate format only, compile the sources with -DWITH_ZLIB. This limits compression and decompression to the deflate format. Only plain and deflated messages can be exchanged, gzip is not supported with this option. Receiving gzip compressed content is automatic, even in the absence of HTTP headers. Receiving deflate compressed content is not automatic in the absence of HTTP headers and requires the flag SOAP_ENC_ZLIB to be set for the input mode to decompress deflated data.

```
    return SOAP_OK;
g
```

13.23 Connecting Clients Through Proxy Servers

When a client needs to connect to a Web Service through a proxy server, set the `soap.proxy_host` string and `soap.proxy_port` integer attributes of the current `soap` runtime environment to the proxy's host name and port, respectively. For example:

```
struct soap soap;
soap_init(&soap);
soap.proxy_host = "proxyhostname";
soap.proxy_port = 8080;
if (soap_call_ns__
```

13.25.2 Creating Client and Service DLLs

Client side DLL serves as the common code which all clients will use to access the server. This DLL consists of the files soapC.cpp

```
#include "stdsoap2.h"
#define PLUGIN_ID "PLUGIN-1.0" // some name to identify plugin
struct plugin_data // local plugin data
{
   int (*fsend)(struct soap*, const char*, size_t); // to save and use send callback
   size_t (*frecv)(struct soap*, char*, size_t); // to save and use recv callback
};
int plugin(struct soap *soap, struct soap_plugin *plugin, void
  struct plugin_data *data = (struct plugin_data*)soap_lookup_plugin(soap, plugin_id); // fetch plugin's local data
  fwrite(buf, len, 1, stderr); // write message to stderr
```