Published in the proceedings of the ACM SIGAPP SAC Conference, 2004 (Embedded Systems track)

Code Generation Techniques for Developing Light-Weight XML Web Services for Embedded Devices

Robert van Engelen*
Department of Computer Science
Florida State University, Tallahassee, FL 32306-4530
engelen@cs.fsu.edu

ABSTRACT

This paper presents specialized code generation techniques and runtime optimizations for developing light-weight XML Web services for embedded devices. The optimizations are implemented in the gSOAP Web services development environment for C and C++. The system supports the industry-standard XML-based Web services protocols that are intended to deliver universal access to any networked application that supports XML. With the standardization of the Web services protocols and the availability of toolkits such as gSOAP for developing embedded Web services, new opportunities emerge to integrate embedded systems into larger frameworks of interconnected applications and systems accessing dynamic resources on the Web ranging from handheld and embedded devices to databases, clusters, and Grids.

Keywords

Networking, XML, Web Services, Embedded Systems

1. INTRODUCTION

Recent developments in Internet protocol standardization have led to the publication of a collection of XML-based protocols that are meant to enhance cross-language and cross-platform interoperability, thereby encouraging systems integration. More specifically, a *Web service*, as defined by the W3C Web Services Architecture Working Group, is "*a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols.*" [22]. This definition is often refined to the use of the Web Services Description Language (WSDL) [23] for describing and advertising Web services, SOAP [4] as the XML-based message packaging format, and HTTP (HyperText Transport Protocol) [12] as the transport protocol.

The ubiquity of HTTP and XML enables a high-level of transport interoperability. The internal logic of an application can be securely exposed to the Internet or local network as a service [20]. Such networked applications can actively participate in various forms of Internet transactions, ranging from simple information services, such as checking flight status information with a cell phone, to more elaborate applications such as Google searches [10], the controlling of "smart" buildings [17], and Grid systems [2, 9] for collaborative scientific research.

However, the complexity and verbosity of XML Web services protocols creates a whole new set of design tradeoffs and issues for developing Web services applications for embedded systems. Embedded systems rarely have enough memory and processing power to run an HTTP Web server, SOAP engine, and XML parser. The verbosity of XML and HTTP increases RAM usage, bandwidth requirements, and operating costs. Current Web services implementations do not adequately address these issues [5, 6, 19].

This paper presents specialized code generation techniques and runtime optimizations for developing light-weight XML Web services for embedded devices. The techniques aim to reduce memory, network, and processing requirements of SOAP/XML over HTTP. The optimizations are implemented in the gSOAP [21] Web services development environment for C and C++, available from SourceForge and included in the IBM alphaWorks Web Services Tool Kit for Mobile Devices (WSTKMD) [11].

The remainder of this paper is organized as follows. Section 2 presents a brief overview of the most widely used systems and protocols for application-level communications. Several Web services implementations for embedded systems are also presented. Section 3 introduces Web services basics. Section 4 presents the gSOAP Web services toolkit followed in Section 5 by the presentation of the optimizations and code generation methods. The paper is summarized with results in Section 6 and conclusions in Section 7.

2. RELATED WORK

Several systems and protocols have been proposed and developed since the early 1980s for inter-application data

*This material is based upon work supported by NSF Grants CCR-9904943, CCR-0105422, CCR-0208892, and by DOE Grant DEFG02-02ER25543. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

SAC '04, March 14-17, 2004, Nicosia, Cyprus

exchange. This section briefly reviews some of the most widely used systems and protocols, and presents Web services development toolkits for embedded systems.

Sun Microsystems' RPC (Remote Procedure Call) compiler generates code that is used by client and server applications to exchange data over a network. The runtime RPC parameter encoding and decoding mechanism is referred to as parameter *marshaling* and *demarshaling*, respectively. The marshaling routines convert application data into XDR (External Data Representation) [13] for transmission. XDR does not support pointer-based data structures, such as graphs. It only describes the most commonly used data-types of high-level languages such as C so that applications written in these languages will be able to communicate.

CORBA is a platform-independent architecture for ORB (Object Request Brokerage) [15]. CORBA uses the IIOP (Internet Inter ORB Protocol) for data transmission between CORBA applications. CORBA's IIOP supports a wide variety of data types that can be specified in the IDL (Interface Description Language). CORBA is a heavy weight product and not very well suitable for embedded devices.

Microsoft's DCOM protocol is similar to IIOP and enables COM objects on different Windows-based systems to communicate. Although DCOM is a platform-independent protocol, it is mainly used within Windows environments.

Sun Microsystems' Java RMI (Remote Method Invocation) automatically marshals objects for communication between Java applications. There is no limit on the type of data objects that can be exchanged, since Java has built-in data *serialization* capabilities. Java supports the serialization of arbitrarily complex data structures such as graphs.

A large number of Web service implementations for various languages are available. A few of these support embedded systems. Note that the "standard" embedded Web servers [3, 18] do not support Web services standards but merely offer an HTTP interface.

The following light-weight Web services implementations are suitable for embedded systems. Most implementations include an (embedded) HTTP Web server. They also typically offer a library-based API (Application Programming Interface) for SOAP/XML packaging. SOAP/XML messages are constructed using a C++ or Java class library.

The micro-services framework [16] considers a subset of the Web services protocols, with a scaled down Web server and SOAP library with limited support for SOAP/XML data types. Only a subset of the SOAP/XML primitive types and compound types is supported.

The eSOAP [8] toolkit for C++ and Java, developed by EXOR International, is a proprietary SOAP implementation for embedded systems with an API based on a class library.

The kSOAP toolkit [7] for Java, developed by Enhydra, runs on embedded devices that support Sun's KVM (Java 2 Micro Edition). The kSOAP toolkit offers a class library for SOAP/XML packaging.

Microsoft .NET compact framework provides a platform-dependent Web services framework for embedded devices. The development of C# Web services is integrated in the Microsoft Visual Studio .NET compilation framework and therefore automated. The .NET framework supports serialization of data objects managed by the CLR (Common Language Runtime). The .NET framework includes the IIS (Internet Information Services) Web server to deploy .NET applications as Web services on the Internet.

The IBM alphaWorks Web Services Tool Kit for Mobile Devices (WSTKMD) for Java, C, and C++ includes the kSOAP and gSOAP toolkits. The gSOAP toolkit [20, 21] is the only Web services development environment that includes a fully automated RPC compiler supporting pure C and C++ Web services applications. The goal of the gSOAP project is to develop an easy-to-use portable Web services software development toolkit that fully automates the development and deployment cycle of efficient C/C++ Web services. The gSOAP compiler selectively generates code to limit application code size and to reduce run-time memory, network, and processing requirements of Web service applications. The gSOAP 2.3 release is WSDL1.1 and SOAP1.1/1.2 compliant and includes an HTTP1.0/1.1 Web server, an XML parser/generator, an RPC compiler, and a WSDL importer (compiler preprocessor). gSOAP 2.3 is portable to most platforms, including Linux, Unix (e.g. AIX, BSD, HP-UX, Irix, Solaris), Cygwin, Windows, Mac OS X, Palm (OS 3,4 and 5), Pocket PC, Symbian, and cell phones. The gSOAP toolkit will be further discussed in Section 4.

3. WEB SERVICES BASICS

Figure 1 shows the layered Web services architecture model.

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

The packaging layer uses the XML-based SOAP protocol. Figure 2 depicts a SOAP 1.1 message transported with HTTP/1.1. A SOAP message consists of an *Envelope* root element, an optional SOAP *Header* to encode meta-information (such as authentication data, signatures, routing points, and transaction tokens), *encoding rules* defining how messages should be processed, and an *RPC representation* in the SOAP *Body* that defines how to represent remote procedure calls and responses, such as the RPC method name and its parameters. The SOAP encoding style uses both scalar types (strings, integers, floats, and so on) and compound types (structures and arrays) that can be used to carry application data. The values of these types appear as XML elements within the method parameters of the SOAP Body. A SOAP *Fault* element (not shown) is used to carry error information to transfer remote exceptions.

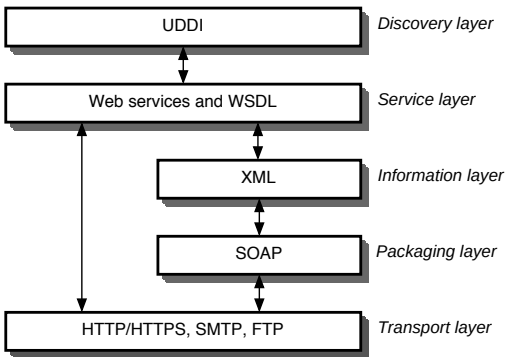

Figure 1: Layered Architecture Model

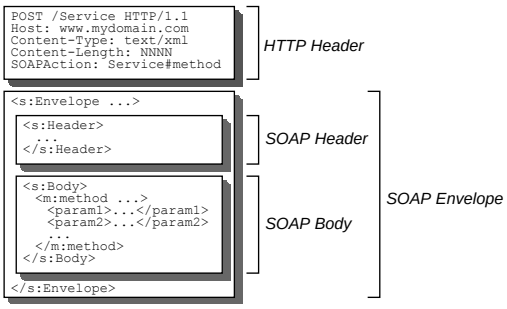

Figure 2: SOAP Message Structure

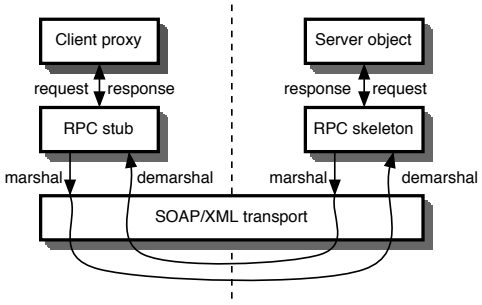

Figure 3: Remote Procedure Calling

The information layer carries the XML-formatted SOAP message. The process of wrapping application data in XML is called *XML serialization*. The mechanisms of XML serialization consists of *XML encoding* to prepare outbound messages for transmission and *XML decoding* upon receiving inbound messages. To establish a SOAP RPC request-response message exchange, a client invokes a local proxy object, see Figure 3. The proxy's RPC stub routine marshals the function parameters in XML, wraps it in a SOAP envelope, and transmits it over the network to be handled remotely, where the reverse process takes place to unwrap the parameters and return a response.

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

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

4. GSOAP

This section gives a brief overview of the gSOAP project. The specialized code generation techniques and runtime optimizations for developing light-weight XML Web services for embedded devices are discussed in Section 5.

4.1 Design Characteristics

The gSOAP Web services toolkit for C and C++ is an open-source development environment for Web services. Important design characteristics include:

- Based on established compiler technologies. The RPC compiler selectively generates code for XML serialization of an application's native C/C++ application data types, including graph-based data structures. This simplifies the development of Web services from a user's point of view [21].

- Light weight, because the RPC compiler generates compact code [20] and the runtime environment of the SOAP/XML engine has a small memory footprint.

- Support for pure C. This is essential for many embedded systems kernels and systems-oriented applications developed in C.

- Enhanced efficiency using XML predictive pull parsing, streaming media techniques, and high-performance latency hiding methods. The performance of gSOAP Web services can surpass the performance of Java RMI and IIOP [19].

- Support for a flexible transport layer supporting synchronous and asynchronous messaging. The communications module implements several callback functions that can be used to replace the standard TCP/IP socket and/or HTTP communications.

- Two-level interface. A service can be defined in WSDL or a gSOAP header file with standard C/C++ syntax for defining services and service parameter data.

- Full support for the basic set of Web services protocols with provisions for building stand-alone (embedded) HTTP/HTTPS Web services.

4.2 Implementation

Figure 4 depicts the development and deployment stages of a gSOAP Web service application. A service application must implement a set of SOAP-compliant RPC functions to expose the service on the Web for remote invocation. This aspect is fully automated. The gSOAP service function interface definitions are specified in a standard C/C++ header file. The service can also be specified with a WSDL document (shaded), which is preprocessed with the gSOAP WSDL importer to produce a header file. This header file

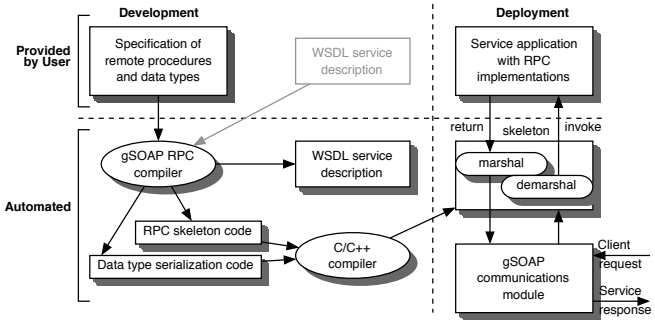

Figure 4: Development and Deployment of a Service

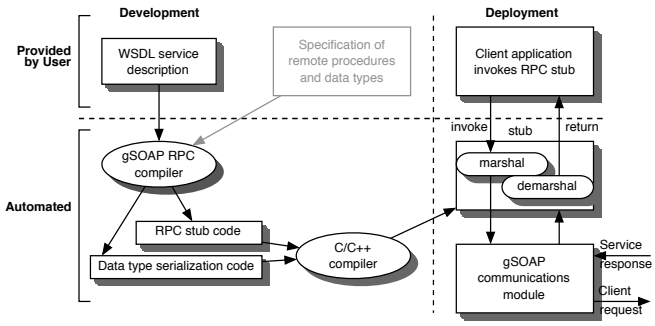

Figure 5: Development and Deployment of a Client

contains the service function prototypes and any additional data type declarations for the function parameters. The header file is compiled with the gSOAP compiler to produce the service RPC skeleton and data type serialization codes. These codes are compiled and linked with the service application to expose the application on the Internet as a Web service. The gSOAP compiler generates a WSDL document describing the service in detail, such as the endpoint location of the service, the remote methods, and the type of parameter data used. This WSDL document can be registered and used by client applications to invoke the service.

Figure 5 depicts the development and deployment stages of a gSOAP client application. The gSOAP WSDL importer and gSOAP compiler are used to parse the WSDL service definitions to create the RPC stub code and XML serialization routines for parameter (de)marshaling by the stub. Optionally, the client stub can be created from the gSOAP header file service definitions. The client application is compiled and linked with the RPC stub and gSOAP communications module to invoke SOAP/XML service functions over the Internet or local network.

4.3 Example

Consider the service interface definitions shown in Figure 6. The first part of the interface defines three XML Web service-specific information items. The gSOAP directive at the first line defines the service name "sensor" and associates the service with an XML namespace qualifier "ns". The namespace qualifier is used as a reference to the service throughout the header file. The second line associates an XML namespace with the service. The third line defines the endpoint for the service, which is the URL that points to the Web service that accepts the SOAP request over HTTP. The service application can be installed in a HTTP Web server or it can be run as a stand-alone gSOAP Web service on a TCP/IP port specified with the endpoint URL.

The sensor response to a poll request consists of a status value, the sensor's most recent sample time, and the sensor's registered temperature. The "tm" structure defined in <time.h> is replicated in the header file to enable XML serialization. The "**volatile**" qualifier indicates that this structure provides (part of) a view of an external type rather than being defined by the header file it is contained in. In this way, the "tm" structure can be serialized even when the fields it contains varies between platforms (the fields shown in the figure are all mandatory). Each "tm" struct field value will be serialized as an XML element within the SOAP/XML sensor response.

```
//gsoap ns service name: sensor
//gsoap ns service namespace: http://.../sensor.xsd
//gsoap ns service endpoint: http://.../sensor
struct ns__sensorResponse
{
  enum sensorStatus { INACTIVE, ACTIVE, FAILURE } status;
  struct tm *sampleTime;
  double sampleTemp;
};
volatile struct tm
{
  int tm_sec; /* seconds (0 - 60) */
  int tm_min; /* minutes (0 - 59) */
  int tm_hour; /* hours (0 - 23) */
  int tm_mday; /* day of month (1 - 31) */
  int tm_mon; /* month of year (0 - 11) */
  int tm_year; /* year - 1900 */
  int tm_wday; /* day of week (Sunday = 0) */
  int tm_yday; /* day of year (0 - 365) */
  int tm_isdst; /* is summer time in effect? */
  char *tm_zone; /* abbreviation of timezone name */
  long tm_gmtoff; /* offset from UTC in seconds */
};
int ns__sensor(char *id, struct ns__sensorResponse *data);
```

Figure 6: Example Service Interface

The last line defines the service RPC function. This function is exposed as a SOAP RPC service method on the Internet. By convention, all but the last parameters of a service function are input parameters. The last parameter is the sensor service response parameter which must be passed by pointer in C or by reference in C++.

The remote "sensor" function is shown in Figure 7. The service function is part of the service application. The service application is compiled and linked with the code generated by the gSOAP compiler for the service interface definitions in the header file that was shown in Figure 6. The "soap" struct contains the gSOAP environment, which is used to make the code reentrant. The sensor "data" structure is populated. The "localtime_r" <time.h> function requires a temporary output buffer. This buffer is allocated with the gSOAP "soap_malloc" function, which returns storage space that is automatically deallocated after the service responded to the client.

The code generated by the gSOAP compiler is linked with the application code shown in Figure 7 to build the service. The application is also linked with the gSOAP runtime engine and Web server to enable the service to run independently (e.g. on a device). The gSOAP HTTP/1.1 server supports keep-alive, chunking, compression, cookie-based state management, authentication, and SSL encryption.

```
int ns__sensor(struct soap *soap, char *id, struct ns__sensorResponse
*data)
{ time_t now = time(NULL); /* assume sensor sample time is now */
  data->sampleTime = localtime_r(&now, soap_malloc(soap,
sizeof(struct tm)));
  data->status = ...; /* INACTIVE, ACTIVE, or FAILURE */
  data->sampleTemp = ...; /* temperature readout */
  return SOAP_OK;
}
```

Figure 7: Example Service Function

```
struct soap *soap = soap_new();
struct ns__sensorResponse data;
if (soap_call_ns__sensor(soap, 0, 0, "ID", &data) == SOAP_OK)
{
  ... = data.status;
  ... = data.sampleTime;
  ... = data.sampleTemp;
}
else
  /* a (remote) fault occurred */
```

Figure 8: Example Invocation

A gSOAP client is build in a similar manner. Consider for example the client-side invocation of a Web service shown in Figure 8. A client application invokes the service using the gSOAP compiler-generated "soap_call_ns__sensor" function (derived from a WSDL or a gSOAP header file), which invokes the service through a stub routine that is compiled and linked with the client application. The stub automatically marshals and demarshals the "id" and "data" parameters using the gSOAP compiler-generated XML serialization code. The "soap_new" function constructs a new runtime environment for the gSOAP engine.

5. GSOAP OPTIMIZATIONS

Additional features were implemented in gSOAP 2.0 and later to limit memory usage, processing time, and bandwidth requirements. These enhancements include improved portability, compact compiler-based code generation methods, and reduced memory overhead with novel streaming techniques for efficient messaging.

5.1 Improved Portability

The gSOAP toolkit supports embedded Linux and WinCE. Additional efforts were made to port gSOAP to the Palm (OS 3,4 and 5), Pocket PC, Symbian, and cell phones. The Palm OS 3 and 4 restrict the code layout by enforcing 64K segments. The gSOAP runtime library (stdsoap2.c) that contains an embedded HTTP Web server and XML parser was split into two parts of about 40K each. The RPC-compiler generated code is placed in a separate segment.

5.2 XML Serialization Optimizations

The gSOAP compiler generates code to serialize native C and C++ data structures, including primitive types, enumerations, structs, classes (with support for polymorphism), pointer-based structures such as graphs, dynamic and fixed-size arrays, and STL containers. gSOAP also provides features to serialize data in customized XML formats. The only types that gSOAP cannot serialize are void pointers, unions, and some STL types. Dynamic arrays are supported through a specific declaration that requires a struct or class with a pointer field to point to the first element in the array and an array size field. In this way, the runtime gSOAP engine can determine array location and size. The gSOAP engine does not augment data types with meta information such as runtime type information. Further improvements were made to support the serialization of application-specific data, fixed library-based data types (using the "**volatile**" qualifier as was illustrated in Section 4.3), and even existing data stored in ROM.

The optimized XML serialization method relies on the well-formed properties of XML. Because XML is a context-free language defined by XML DTDs (Document Type Definitions) or XML schemas, which define grammar-like rules for parsing and validating XML content, XML can be parsed with a parser that is derived from these definitions. Tools such Bison and Yacc can in principle be used to construct an optimized parser from the XML DTD/schema grammar. However, a light-weight recursive descent parser suffices, because XML (as defined by DTDs or schemas) is an LL(1) language. A recursive descent parser is a predictive parser that rejects input when the input does not correspond to a fixed set of tokens that are expected on the input based on the LL(1) properties of the language [1].

The gSOAP RPC compiler generates predictive parsers for XML. The generated code is a predictive parser for the specific XML contents associated with a Web service invocation. The XML content associated with a Web service invocation is defined in the XML schema types part of a WSDL service definition[1]. The predictive parser generated by the gSOAP RPC compiler also converts XML into C/C++ application data (similar to the semantic rules of an augmented grammar). This decoding constructs application data without requiring an elaborate post-processing phase for data conversion after decoding, except that forward references in XML have to be resolved later by back-patching pointers in graph-like data structures [20]. The recursion of the parser is carefully implemented by limiting the amount of stack-allocated data. A gSOAP environment is passed to each function call. The environment ensures that the code is reentrant and provides temporaries such as buffers that can be re-used through the recursive calls.

The optimized serialization algorithm for XML encoding uses two phases. The first phase determines which data components belong to the data structure and which pointers are used to reference them. This phase is only relevant for pointer-based data structures. The second phase emits the XML encoded form of the entire data structure, with all sub-components of the structure serialized recursively.

Phase 1: traverse the data structure graph by visiting each node and by following the pointer references to all sub-nodes. For each pointer that was followed to a sub-node, store the pointer's target address in a hash table together with an identification of the data type referenced by the pointer. The hash table key is a triple of <*PtrLoc*,*PtrType*,*PtrSize*>, where *PtrLoc* is the pointer's target address, *PtrType* is the type of data referenced by the pointer, and *PtrSize* is the size of the object in bytes, which is statically determined with "sizeof". The *PtrSize* of dynamic arrays is computed from the array size and element size, where dynamic arrays are defined with a struct/class containing a pointer field and size field. Node pointers are only followed through to visit sub-nodes when the key <*PtrLoc*,*PtrType*,*PtrSize*> is not already contained in the hash table. When the key is already contained in the hash table, then the hash table entry is marked *RefType*=multi to indicate that a multi-referenced sub-node has been found. Entries in the hash table are marked *RefType*=embedded when a data element that

[1]gSOAP can also parse arbitrary XML documents with the XML DOM parser that is included in the gSOAP 2.3 release.

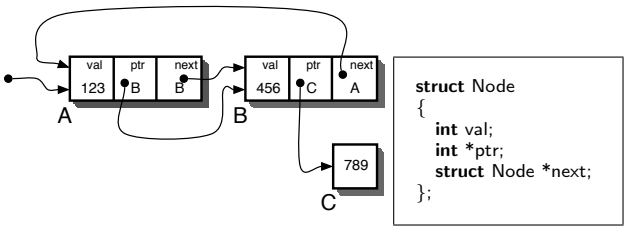

Figure 9: Data Structure Graph and Node Structure Declaration

```
<Node id="_1">
  <val>123</val>
  <ptr href="#_2"/>
  <next>
    <val id="_2">456</val>
    <ptr>789</ptr>
    <next href="#_1"/>
  </next>
</Node>
```

Figure 10: Serialized XML Output of the Data Structure Graph

is pointed to is embedded in larger structure, such as a field of a struct or class, or an element of an array. It is noteworthy to mention that a hash table entry is created at run time only for each pointer in a pointer-based data structure. No additional space is required to serialize non-pointer-based structures.

Phase 2: emit the XML encoded data by visiting each node in the data structure graph and by following the pointer references to the sub-nodes for recursive serialization. Multi-referenced nodes (those whose hash table entry is marked *RefType*=multi) are serialized separately, as required by SOAP 1.1 encoding. The serialization settings can be changed to override SOAP 1.1 encoding to serialize data in a more generic XML format, such as SOAP 1.2 encoding. In that case, also special care is taken to serialize data elements that are referenced by pointers and which are embedded within structs or arrays (that is, the hash table entry for these elements are marked *RefType*=embedded). The embedded property of data elements affects the id-href element cross referencing produced in the encoded form of XML. The cross referencing produced ensures that the receiving side of the XML message can accurately reconstruct the serialized data structure from the XML content alone.

The two-phase serialization is illustrated with the example data structure shown in Figure 9. The structure consists of three nodes, two structs located at addresses "A" and "B", and a node that contains a single integer value stored at address "C". The data type declaration of the node struct with the "val", "ptr", and "next" fields is shown in Figure 9.

The serialization starts at the root struct stored at location "A". The first phase consists of a pass over the entire data structure to collect the properties of the pointers used in the data structure graph and to store these in the hash table:

ID	*PtrLoc*	*PtrType*	*PtrSize*	*PtrCount*	*RefType*
1	A	Node	12	> 1	multi
2	B	int	4	1	embedded
3	B	Node	12	1	single
4	C	int	4	1	single

Each entry has a unique index *ID*, the hash table key which is the triple <*PtrLoc*,*PtrType*,*PtrSize*> with target pointer address *PtrLoc* and target type pointed to *PtrType*, an indication of the number of references made to this target address, *PtrCount*, which is either "1" or "> 1", and the type of the reference *RefType*, which is either "single", "multi", or "embedded".

The serialized XML output is shown in Figure 10. The root node is serialized with "`id="_1"`", because it is multi-referenced. The second struct at location "B" is serialized in XML as a nested element of the first node struct, because it has only a single reference. Note that the "ptr" field in the first struct points to the "val" field in the second struct, which is at location "B". Because the "val" field is embedded within a struct, the "ptr" is serialized with a forward pointing "`href="#_2"`" attribute. This ensures that the receiving side can decode the XML and backpatch the "ptr" pointer field to point to the "val" field after the contents of the second struct are decoded that contains the value of "val". The "ptr" field in the second struct points to a single-referenced integer located at "C". The XML serialized value is placed directly in an "`ptr`" element without an "`href`" attribute, because it is a single reference. The SOAP encoding style can be used with gSOAP to produce SOAP compliant XML, which requires the two multi-referenced elements to be placed in a separate set of elements.

The gSOAP compiler-generated predictive parser decodes the contents to the original data structure graph. The parser takes special care in handling the XML "`id`" and "`href`" attributes, which resemble references. When the data structure is (re)constructed, unresolved references are kept in a hash table. When the target objects of the references have been parsed and the data is allocated in memory, the unresolved references are replaced by pointers. In effect, the unresolved pointers in the Node structures are back-patched with pointer values to link the separate parts of the (cyclic) graph structure together.

XML parsing is further optimized using look-aside buffers to store XML attribute names with their corresponding values. The look-aside buffers ensure that dynamic allocation is minimized while parsing XML attributes for decoding. XML element tag names are buffered in a single pre-allocated buffer to conserve memory.

5.3 XML Streaming

A recent study on XML Web services with mobile devices on wireless networks [14] concluded that SOAP/XML suffers from a performance penalty and is slower compared to IIOP and Java RMI. However, the SOAP/XML implementations considered by the authors in their study are suboptimal. Implementations that utilize XML DOM (Document Object Model) for example, must construct a DOM tree for an outbound XML message. Likewise, an XML DOM parser is used to construct the DOM tree of an inbound message. The application data can only be extracted from the DOM tree after the entire message has been received, i.e the op-

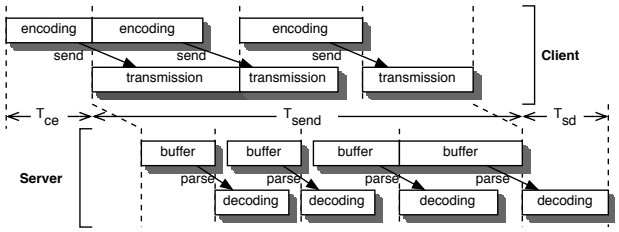

Figure 11: HTTP Chunking and XML Streaming

posite of streaming. Therefore, the use of an XML DOM or another intermediate data representation can incur a significant overhead in realistic Web service invocation scenarios.

The gSOAP communications module was augmented to support HTTP chunking and additional XML pull parsing techniques to achieve streaming. With chunking, the HTTP header (Figure 2) does not need to indicate the message content length (which requires the buffering of the entire message to determine its length[2]), but indicates that the body of the HTTP message is chunked in blocks. The encoding and decoding of application data in XML is supported with gSOAP's XML serializers that are generated by the gSOAP compiler. These serializers directly operate on application data to transmit the data in XML and to parse XML and convert it back into application data without any intermediate forms of XML such as a DOM tree or SOAP data structure. Therefore, XML is not used as a persistent document format. It simply forms the essential standardized format to exchange platform-independent information and the proper optimization techniques such as streaming can greatly enhance the performance of XML communications.

Figure 11 illustrates the streaming of SOAP/XML request messages from a client to a server. The encoding of application data to XML takes place by the gSOAP engine while the message is transferred in chunks of XML. At the sending side, each chunk is encoded and then transmitted at the same time the next chunk is being encoded. At the receiving side, XML messages are parsed and decoded as soon as data arrives in a buffer. Note that the size of the data received in this buffer does not need to correspond to the size of a chunk, because parsing and decoding can commence as soon as part of the buffer is filled by a receive operation.

The optimal chunk size depends on the network bandwidth, processor speed, and average message length. A fast network or processor requires a larger chunk size. A chunk size of 32K to 64K appears to be a good choice for most desktop systems. Performance of single-CPU platforms degrades significantly for a chunk size below 32K. The chunk size should also not be too small to avoid sending small TCP/IP packets which can severely reduce throughput. Chunk sizes above 64K may increase the performance by an additional 10% on high-speed networks. However, chunk size will be limited by the embedded systems' memory constraints and network capabilities.

5.4 Compression

The gSOAP engine was extended with runtime compression using Zlib. Compression may improve performance (and reduce the size of the verbose XML messages), but software compression methods such as the Zlib library generally add a significant overhead to the message encoding and decoding time [19]. Streaming compression with HTTP compression methods, such as gzip, deflate, or compress, combined together with HTTP chunking may be used to reduce this overhead. However, V90 modems commonly used with dialup connections for example, already apply packet-based compression, which eliminates the need for software compression.

[2]gSOAP uses a separate SOAP encoding phase to determine the HTTP header content length, see [20] for more details

5.5 Binary Data Transfer Optimizations

To implement efficient binary data transfers, the DIME (Direct Internet Message Encapsulation) protocol was integrated into the gSOAP transport. DIME transport attachments carry binary data associated with a SOAP/XML request or response. The DIME format results in better performance, lower processing cost, and reduced code size compared to MIME attachments. In addition, DIME attachments can be streamed to achieve a higher throughput. This feature allows file contents to be transmitted as an attachment to a SOAP/XML request and response without requiring the file to be resident in memory. Therefore, data duplication is not necessary. A file is incrementally send in chunks from disk and attachments are saved in chunks to disk. This streaming property is very useful for small scale systems such as mobile devices and embedded systems.

6. RESULTS

The size of a Web service application is determined by the set of data types that the service supports. The table below lists the code sizes for a select set of applications:

Application	*Code Size (KB)*
gSOAP Web server and SOAP engine	77
GetQuote client (subset of features)	83
GetQuote client	85
Google API client	164
Interop A server	300
Interop B server	188
Interop C server	155

Code size was measured on a Red Hat Linux 2.4 system, running on a P3, and the applications were compiled with gcc 2.95.3 -O2. The gSOAP Web server and SOAP engine are essential components and statically linked with the gSOAP clients and servers. No other libraries except the standard C library and socket library were linked with the applications. The Zlib and OpenSSL libraries were not used for these tests. The GetQuote client application is a C program that retrieves a stock quote from a server. The code size of the minimal configuration and typical configuration of GetQuote are shown. The Google API client application is a C program that dynamically invokes the Google search engine using the Google API [10] and retrieves the Google search results in XML. The [24] interoperability testing site allows Web service toolkit projects to install interoperability servers. These servers echo various types of data structures back to test client applications to verify compliance with the SOAP 1.1 protocol. The interoperability class A is designed to test an extensive set of primitive types, structs, and one dimensional arrays. Interoperability class B is designed to test compound types such as structs, one dimensional, and two dimensional arrays in a number of combinations. Interoperability class C test the SOAP Header compliance.

The table below lists the total maximum memory footprint of the applications on the Linux system:

Application	*Malloc (KB)*	*Footprint (KB)*
GetQuote client	1.5	546
Google API client	2.4	1,004

The *Malloc* column lists the total amount of dynamically allocated data. The *Footprint* column lists the footprint of the application with the default stack and heap size settings under Red Hat Linux. The Google API search was performed on the string "`embedded systems`". The interoperability servers are multi-threaded. A multi-threaded server processes requests simultaneous thereby increasing the footprint as a function of the load. Therefore, the interoperability servers are not shown in the table with the footprint statistics.

These results show that the kernel gSOAP system is small in terms of code size and total memory footprint. Applications that include a wide variation of data types, such as the interoperability servers. have not shown to cause the gSOAP compiler to generate excessive code. The memory footprint of the applications on the P3-based Linux machine shows a reasonably sized pre-allocated stack and heap space. However, the dynamic memory allocation statistics also show that the impact of the client applications on the memory utilization is limited, which is a desirable property for running Web service applications on embedded systems.

7. CONCLUSIONS

The gSOAP toolkit was augmented with many new features and optimizations to support embedded systems. The optimizations further reduce memory requirements, network bandwidth requirements, and processor demands to meet the stringent constraints of embedded devices. These improvements include enhanced portability, new code generation methods for XML serializers, XML streaming techniques, message compression, and support for efficient binary data transfers with (streaming) DIME attachments. Results indicate that the code size and memory footprint are small, even for client and server applications that have to exchange relatively complex data structures.

The gSOAP system is part of a number of larger software development packages for embedded systems, such as IBM WSTKMD, and WindRiver and OpenWave products. The system has been used for developing (mostly proprietary) Web services applications for handheld devices and embedded systems.

8. ACKNOWLEDGMENTS

The author would like to thank the IBM Emergent Technologies team, and Bob Goodman in particular, for valuable discussions and help in porting gSOAP to the Palm OS and Symbian devices.

9. REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, Reading MA, 1985.

[2] G. Aloisio, M. Cafaro, D. Lezzi, and R. van Engelen. Secure web services with Globus GSI and gSOAP. In *in the proceedings of EUROPAR 2003 conference*, 2003.

[3] G. Borriello and R. Want. Embedded computation meets the World Wide Web. *Communications of the ACM*, 43(5):59–66, May 2000.

[4] D. Box et al. Simple object access protocol 1.1, 2000. http://www.w3.org/TR/SOAP.

[5] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In *proceedings of the 11th IEEE International Symposium on High-Performance Distributed Computing*, 2002.

[6] D. Davis and M. Parashar. Latency performance of SOAP implementations. In *2nd IEEE International Symposium on Cluster Computing and the Grid*, 2002.

[7] Enhydra. kSOAP. http://ksoap.enhydra.org/.

[8] EXOR International. eSOAP. http://www.embedding.net/eSOAP/.

[9] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the Grid: An Open Grid Services Architecture for distributed system integration. Technical report, the Globus project, 2002. http://www.globus.org/research/papers/ogsa.pdf.

[10] Google. Google Web API. http://www.google.com/apis/.

[11] IBM alphaWorks. Web services tool kit for mobile devices, 2002. http://www.alphaworks.ibm.com/tech/wstkMD.

[12] IETF. HTTP 1.1 specification. www.ietf.org/rfc/rfc2616.txt.

[13] IETF. XDR specification. www.ietf.org/rfc/rfc1014.txt.

[14] M. Laukkanen and H. Helin. Web services in wireless networks — what happened to the performance? In *International Conference on Web Services (ICWS)*, pages 278–284, Las Vegas, 2003.

[15] OMG. CORBA component model. http://www.omg.org.

[16] I. Pratistha, N. Nicoloudis, and S. Cuce. A micro-service framework on mobile devices. In *International Conference on Web Services (ICWS)*, pages 320–325, Las Vegas, 2003.

[17] D. Snoonian. Smart buildings. *IEEE Spectrum*, 40(8):18–23, 2003.

[18] D. Tennenhouse. Proactive computing. *Communications of the ACM*, 43(5):43–50, May.

[19] R. van Engelen. Pushing the SOAP envelope with web services for scientific computing. In *proceedings of the International Conference on Web Services (ICWS)*, pages 346–352, Las Vegas, 2003.

[20] R. van Engelen and K. Gallivan. The gSOAP toolkit for web services and peer-to-peer computing networks. In *2nd IEEE Int'l Symposium on Cluster Computing and the Grid*, 2002.

[21] R. van Engelen, G. Gupta, and S. Pant. Developing web services for C and C++. *IEEE Internet Computing*, pages 53–61, March 2003.

[22] W3C. Web services architecture requirements. http://www.w3.org/TR/wsa-reqs.

[23] W3C. WSDL specification. http://www.w3.org/TR/wsdl.

[24] White Mesa. White mesa interop lab. http://www.whitemesa.com/interop.htm.