Published in the proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002), May 21-24, 2002, Berlin, Germany.

The *gSOAP* Toolkit for Web Services and Peer-To-Peer Computing Networks

Robert A. van Engelen[*] and Kyle A. Gallivan[†]
Department of Computer Science and School of Computational Science and Information Technology
Florida State University
Tallahassee, FL 32306-4530
{engelen,gallivan}@cs.fsu.edu

Abstract

This paper presents the gSOAP *stub and skeleton compiler. The compiler provides a unique SOAP-to-C/C++ language binding for deploying C/C++ applications in SOAP Web Services, clients, and peer-to-peer computing networks.* gSOAP *enables the integratation of (legacy) C/C++/Fortran codes, embedded systems, and real-time software in Web Services, clients, and peers that share computational resources and information with other SOAP-enabled applications, possibly across different platforms, language environments, and disparate organizations located behind firewalls. Results on interoperability, legacy code integration, scalability, and performance are given.*

1. Introduction

Many recent efforts in distributed computing are aimed at developing general-purpose distributed computing infrastructures, providing integrated security, availability, scalability, reliability, and manageability for general distributed computing applications. Examples of infrastructures exploring general distributed computing are Charlotte [1], Javalin/Javalin++ [13, 14], HARNESS [2], Legion [10], and Globus [7]. These infrastructures address the technologies needed to build computational grids [8]. Grids are persistent environments that enable software applications to integrate instruments, displays, computational and information resources that are managed by diverse organizations.

A number of distributed computing infrastructures embrased Java as the programming language of choice. Java has many features that make it desirable for distributed computing. Java's low performance can sometimes be increased through proprietary packages and compilation techniques [12]. However, the integration of system software, small-scale embedded systems, and real-time software remains problematic. Because software development has become more expensive than the cost of technology, porting (legacy) codes to Java is not cost effective. The use of the Java native interface can leverage production cost somewhat, but requires the writing of application wrapper routines by hand which is costly and error prone.

A more recent development is the Simple Object Access Protocol (SOAP) [3]. SOAP is a versatile message exchange format that is simple and light-weight. The XML-based protocol is language and platform neutral, which means that information sharing relationships can be initiated among disparate parties, across different platforms, languages and programming environments. SOAP is not a competitive technology to component systems and object-request broker architectures such as the CORBA component model [15] and DCOM, but rather complements these technologies. CORBA, DCOM, and Enterprise Java [17] enable resource sharing within a single organization while SOAP technology aims to bridge the sharing of resources among disparate organizations possibly located behind firewalls. SOAP applications exploit a wire-protocol (typically HTTP) to communicate with Web Services to retrieve dynamic content. For example, real-time stock quote information of a stock portfolio can be graphed on the display of a cell phone or can be analyzed within a spreadsheet program running on a desktop computer. This allows real-time "what-if" scenarios and enables the development of agents that access real-time information [21]. Other examples are the visualization of factory processes on PDAs, control and visualization of large-scale simulations from a desktop computer, the sharing of laboratory results using cell phones, remote database access, and science portals.

This paper presents the *gSOAP* software development kit (SDK) [20]. The *gSOAP* stub and skeleton compiler enables (legacy) C/C++ and Fortran applications (via C-to-Fortran bindings) to share computational resources and information with other applications, possibly across different platforms, language environments, networks, and organizations. As a result, scientific applications, embedded systems, and real-time software can interoperate as clients, ser-

[*]Supported in part by NSF grants CCR-0105422 and CCR-9904943
[†]Supported in part by NSF grant EIA-0072043

vices, or peers. The stringent memory and response time requirements by these systems require efficient and specialized (de)marshalling algorithms. Especially important for numerical applications and libraries to keep the optimized data structures intact. Other SOAP C++ toolkits adopt class libraries for SOAP-specific data types. The libraries use pointers, lists, and heap space extensively. Such implementations are undesirable for legacy applications and embedded and real-time systems because of the recoding effort and the resulting indirect memory accesses, heap storage, and performance degradation.

The remainder of this paper is organized as follows. Section 2 discusses the key features of SOAP for Internet and peer-to-peer computing. Section 3 introduces the *gSOAP* compiler design, implementation, and use. Preliminary results are presented in Section 4, including interoperability, legacy code integration, and scalability and performance tests. Finally, Section 5 summarizes the conclusions.

2. The Simple Object Access Protocol

This section discusses SOAP and presents a brief overview of SOAP APIs and SDKs.

2.1. Interoperability

SOAP is a language- and platform-neutral RPC protocol that adopts XML as the marshalling format. SOAP applications typically use the firewall-friendly HTTP transport protocol. These and other key interoperability features of SOAP are summarized below:

Ubiquity. The SOAP protocol and its industry-wide support promises to make services available to users anywhere, e.g. in cellphones, pocket PCs, PDAs, embedded systems, and desktop applications.

Simplicity. SOAP is a light-weight protocol based on XML. An example of a simple SOAP service is a sensor device that responds to a request by sending an XML string containing the sensor readout. This device requires limited computing capabilities and can be easily incorporated into an embedded system.

Services. SOAP Web Services are units of application logic providing data and services to other applications over the Internet or intranet. A Web Service can be as simple as a shell or Perl script that uses the Common Gateway Interface (CGI) of a Web server such as Apache. A web service can also be a server-side ASP, JSP, or PHP script, or an executable CGI application written in a programming language for which an off-the-shelf XML parser is available.

WSDL. The Web Service Description Language (WSDL) is an XML format for describing network services as abstract collections of communication endpoints capable of exchanging structured information. The platform- and language-neutral WSDL descriptions published by Web Services enable the automatic generation of SOAP stubs for the development of clients within a specific programming environment. The language-specific stubs can be used to invoke the remote methods of the Web Service, see Figure 1.

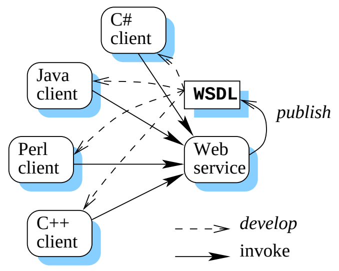

Figure 1. Web Service, WSDL, and Clients

UDDI. The Universal Description, Discovery, and Integration (UDDI) specification provides a universal service for registry, lookup, discovery, and integration of worldwide business services [19]. WSDL descriptions complement UDDI by providing the abstract interface to a service.

Transport. A SOAP message can be sent using HTTP, SMTP, TCP, UDP, and so on. A SOAP message can also be routed, meaning a server can receive a message, determine that it is not the final destination, and route the message elsewhere. During the routing, different transport protocols can be used.

Security. SOAP over HTTPS is secure. The entire HTTP message, including both the headers and the body of the HTTP message, is encrypted using public asymmetric encryption algorithms. SOAP extensions that include digital signatures are also available, see W3C note [4]. Single sign-on (authentication) and delegation mechanisms [8] required for Grid computing can be easily built on top of SOAP.

Firewalls. Firewalls can be configured to selectively allow SOAP messages to pass through, because the intent of a SOAP message can be determined from the message header.

Compression. HTTP1.1 supports gzipped transfer encodings, enabling compression of SOAP messages on the fly.

Persistence. Long-term data persistence using NFS or database storage is only useful if the protocols and content encoding formats also persist in the form of meta-data. Applications are frequently upgraded, or worse, support is terminated. Software upgrades can make it difficult to extract data stored in outdated formats and often this means that information is lost. Because SOAP is based on XML and XML is often heralded as “self-describing”[1], the SOAP message and its content can be easily made persistent.

[1]XML documents contain meta-data such as type information. XML schemas formally describe the structure and content of XML documents.

Transactions. SOAP provides transaction-handling capabilities through the SOAP message header part. The transaction-handling capabilities allow a state-full implementation of a Web Service by a server-side solution utilizing local persistent storage media.

Exceptions. SOAP supports remote exception handling.

The possible disadvantages of SOAP are:

GC. The absence of mechanisms for distributed garbage collection (GC) and the absence of objects-by-reference.

Floats. Floats and doubles are represented in decimal (text) form in XML, which can possibly contribute to a loss of precision. Other SOAP encodings such as hexBinary and Base64 can be used to encode e.g. IEEE 754 standard floating point values, but this may hamper the interoperability of systems that use other floating point representations.

2.2. SOAP Toolkits for Internet Computing

A large number of SOAP toolkits are available for different programming languages and platforms, see e.g. [11]. Toolkits range from simple APIs to elaborate SDKs for SOAP client and server development. Examples are SOAP::Lite for Perl, Apache SOAP for Java, the .NET framework SDK, and the *gSOAP* toolkit for C and C++.

The use of SOAP for simple applications does not necessarily require an elaborate SDK for the application to participate in a distributed computing infrastructure. A SOAP application can be as simple as a script that prints an XML string. Consider for example the shell script shown in Figure 2. The `echo` command sends an XML-formatted request for a sensor readout to a simple SOAP remote method dispatcher utility `dispatch`. This utility simply forwards the string to a SOAP service connected to `dbsrv.cs.fsu.edu` port 18080 by opening a socket, passing the string on to the service, and by returning the response to the standard output stream. In Figure 2, the `deser` utility applies an XSLT transformation to the SOAP XML response to emit the result in a readable text form. XSLT transformations can be used to transform XML into any format, including plain text, HTML, and PDF.

Script languages such as Perl provide flexible SOAP implementations that typically support dynamic stub generation by utilizing WSDL descriptions. Figure 3 depicts a Perl SOAP client implemented with SOAP::Lite that utilizes the WSDL description of the XMethods Delayed Stock Quote service to dynamically create a `getQuote` proxy and stub to request a stock quote.

```
#!/bin/sh
echo "<e:Envelope xmlns:e=... xmlns:n=...><e:Body>\
<n:getReadout><sensor>$1</sensor></n:getReadout>\
</e:Body></e:Envelope>"\
| dispatch "dbsrv.cs.fsu.edu:18080" | deser
```

Figure 2. A Shell-Based SOAP Client

```
use SOAP::Lite;
print SOAP::Lite
 -> service(
     'http://www.xmethods.net/sd/StockQuoteService.wsdl')
 -> getQuote('AOL');
```

Figure 3. A Perl SOAP Client

Most Java SOAP toolkits offer run-time stub generation facilities using dynamic type inspection and/or provide compile-time stub generation. The lower response time of a client that uses a pre-compiled stub improves the overall quality of service of the client. Precompiling the stub routines saves the overhead for generating these routines at run time via dynamic type inspection. An example Apache SOAP 2.2 Java client that accesses the Delayed Stock Quote Service has about 40 lines of Java code (not shown).

SOAP C++ toolkits adopt class libraries for marshalling SOAP data types with an XML parser and generator. Each SOAP data type, which is essentially an XML schema type, has a corresponding class in the library. The library approach forces a user to adapt the application logic to these libraries or the user has to implement wrappers by hand that copy the application-specific data structures into the SOAP-specific data structures and vice-versa.

2.3. SOAP Peer-To-Peer Computing

SOAP does not enforce a strict client-server relationship but does not endorse a specific peer-to-peer (P2P) architecture either. A platform-independent SOAP P2P computing infastrucure can be build using JXTA [18] and the .NET framework.

The use of a shell for "on the fly" distributed computation with pipes to connect applications as shown in Figure 2 is somewhat similar to the JXTA shell. However, in contrast to Unix pipes, JXTA pipes are bidirectional allowing applications to be peers with respect to eachother.

The MS .NET framework SDK enables the development of P2P infrastructures with UDDI, WSDL, and SOAP clients and Web Services developed in languages such as Visual Basic, C#, C++, and Haskell. The .NET architecture includes the Common Language Runtime (CLR) for managing objects. The CLR can perform SOAP message exchange as part of the serialization capabilities of objects managed by the CLR. This solution is platform-specific and the strict use of managed objects on the heap has an impact on memory use and overall performance. Furthermore, Visual C++ applications have to use an interface to the CLR to create and use managed objects which adds a whole new layer of complexity to pure C++ applications.

3. The *gSOAP* Stub and Skeleton Compiler

This section motivates the design of the *gSOAP* stub and skeleton compiler and briefly presents its implementation.

3.1. Design Characteristics

The critical *gSOAP* compiler design characteristics are:

Precompiling (De)Marshalling Routines. The stubs, skeletons, serialization, and deserialization routines are pre-compiled to minimize dynamic type inspection. Deserialization deals with SOAP "forward" references by tracking unresolved pointers. Serialization is a two-stage process to comply to SOAP's multi-referenced object encoding rules.

Support for Native Data Types. The pre-compiled marshalling routines serialize and deserialize native C/C++ and user-defined data types and this data is not extended with additional information such as tags. As a result, restructuring compiler techniques can be used to compile and optimize an application's kernel routines together with the marshalling routines. For example, a restructuring compiler can optimize data placement in memory to reduce memory access latencies [16]. Also cache-conscious data placement strategies can be used to improve memory access [5].

Minimizing Memory Operations. To avoid data copying overhead, serialization and deserialization operations are performed *in situ* on the application's native data structures that are static, stack, and/or heap allocated.

Mimimizing Memory Use. Stand-alone client and server executables have a small memory footprint (typically less than 150K). This enables deployment of SOAP clients and services in small-scale embedded systems. The commonly used technique to buffer the whole output message to determine the HTTP message length is expensive for small devices such as PDAs. Instead, the message length is determined in a separate serialization pass (two-stage "count and send" serialization). Additional memory overhead is limited to the use of a hash table required for the determination of multireferenced objects. The size of the hash table corresponds to the number of pointers within the marshalled data structures.

Efficient XML Parsing. *gSOAP*'s runtime library includes a customized parser that parses XML on demand without keeping parts of the XML document in memory.

Preservation of Structure. When a data structure is encoded in SOAP and decoded on the receiving side it is an exact copy of the structure of the original. The copy, however, may occupy a different memory region, including static/stack/heap allocations, and therefore pointers within the copy may use different address values than the original. To preserve structure, SOAP multireferenced object encoding is used to encode arbitrary (cyclic) graph structures.

Legacy Application Integration. The use of pre-compiled marshalling routines for native C/C++ and user-defined data types enables the integration of C and C++ legacy applications within SOAP clients, services, and peers. This also enables the integration of Fortran legacy applications within SOAP clients and services through the use of existing C-to-Fortran bindings provided by the Unix linker or through MS Windows DLLs.

Platform Independence. The compiler generates platform-independent C and C++ source code for the stubs, skeletons, and marshalling routines. SOAP clients and services run on Linux, Unix, MS Windows 98/2000/NT/XP/CE, PocketPC, and embedded systems.

3.2. Implementation

The stubs of the remote methods to be invoked by a SOAP client are generated by a utility that translates the WSDL service description of a Web Service into C and C++ declarations that are stored in a standard header file. This translation makes the remote method proxy interface transparent to a user. The *gSOAP* compiler executable, `soapcpp`, processes the C/C++ declarations and generates C and/or C++ source code stubs for integration in a client.

This unique aspect of *gSOAP*'s SOAP-to-C/C++ language binding is illustrated with an example SOAP client. The client program prints the stock value of the stock ticker symbol provided as an argument. The `quote.h` header file shown in Figure 4 is produced from the XMethods Delayed Stock Quote service WSDL. This function prototype specifies all of the necessary details for `soapcpp` to generate the stub. The string and float primitive types of the remote method parameters are encoded and decoded in SOAP as standardized XML schema types (i.e. xsi:type="xsd:string" and xsi:type="xsd:float"). The client program is shown in Figure 5. The client uses the proxy `soap_call_ns__getQuote` generated by `soapcpp` from `quote.h`. The source code includes the `namespaces` table with XML namespace URIs. The service namespace URI is `urn:xmethods-delayed-quotes` which is bound to the `ns` prefix.

The implementation of a SOAP Web Service is initiated with the writing of an interface description in a C or C++ header file. The header file contains the declaration of the remote methods and the data structures used by the remote method parameters. The `soapcpp` compiler generates the skeleton and marshalling routines from the header file. Also a WSDL service description is generated by `soapcpp`. An example Web Service is presented in Section 4.1.

```
int ns__getQuote(char *symbol, float &result);
```

Figure 4. `quote.h`

```
const char endpt[] = "http://services.xmethods.net/soap";
main(int argc, char **argv)
{ float q;
  if (!soap_call_ns__getQuote(endpt, "", argv[1], q))
    cout << q;
}
struct Namespace namespaces[] = {
{"SOAP-ENV","http://schemas.xmlsoap.org/soap/envelope/"},
{"SOAP-ENC","http://schemas.xmlsoap.org/soap/encoding/"},
{"xsi",     "http://www.w3.org/2001/XMLSchema-instance"},
{"xsd",     "http://www.w3.org/2001/XMLSchema"},
{"ns",      "urn:xmethods-delayed-quotes"},
{NULL,      NULL} }
```

Figure 5. A C++ SOAP Client

3.3. Marshalling and Demarshalling

The `soapcpp` compiler generates (de)marshalling routines that are SOAP 1.1 and mostly SOAP 1.2 compliant. The C/C++ data types are encoded and decoded as SOAP XML schema types as follows:

Basic Types. The primitive C types (including `bool`, `char*`, `wchar_t*`, and `time_t`) are encoded as built-in primitive XML schema types, such as xsd:int, xsd:double, and xsd:string. A `typedef` construct is used to inform the `soapcpp` compiler how to store XML schema types, e.g:

```
typedef double xsd__decimal;
```

This allows (legacy) applications to use doubles with no change, while the doubles are encoded and decoded as e.g. xsd:decimal schema types.

Enumerations. C enumerations are recognized by the `soapcpp` compiler and (de)serialized with symbolic names as XML schema enumeration types.

Structs. The ComplexType XML schema type is used for data structures that resemble structs with fields in SOAP.

Classes. Like structs, the ComplexType XML schema type is used for encoding classes instances. Only single class inheritance is supported due to SOAP encoding constraints. Serialization and deserialization methods are automatically added by `soapcpp` to class definitions. The methods recursively encode/decode all fields of a class. Dynamic method binding is used to serialize derived class instances at run-time. This enables "black-box" clients and services that operate on derived class instances.

Pointers. Pointers are not explicitly part of a SOAP payload. However, SOAP supports multi-referenced objects and nil objects in XML which allows serialization of pointer-like structures such as lists, trees, and arbitrary (cyclic) graphs. Such structures keep their structural identity. The compiler assumes that all pointers used in an application point to a single object in memory and `void` pointers are not supported. Applications that use pointers to point to multiple objects need to use dynamic arrays instead to reveal the number of elements refered to by the pointer to the serializers, see dynamic array encoding below.

Fixed-Size Arrays. Fixed-size C/C++ arrays are marshalled as SOAP-ENC:Array types.

Dynamic Arrays. Many C/C++ applications use pointers to arrays. This poses a problem for the serializers to get the size of the array pointed to. The compiler recognizes a special data structure for dynamic arrays that are declared as a struct or class with a pointer field and a size field, e.g.

```
struct ArrayOfInt { int *__ptr; int __size; } A
```

declares an array `A` of ints which is encoded as a SOAP-ENC:Array with arrayType xsd:int[]. Multi-dimensional arrays can be declared in a similar way.

Compound Types. A SOAP compound type is a struct/class, array, or a list of unordered elements. Such lists are supported through the declaration of a dynamic array with a namespace prefix for the struct/class name, e.g.

```
class ns__vector { X *__ptr; int __size; } V
```

declares a vector `V` of `X` encoded as a ns:vector schema type.

Special Types. The xsd:base64Binary, xsd:hexBinary, and SOAP-ENC:base64 XML schema types are useful for transmitting raw binary data such as images. These types are declared as a dynamic array of type `unsigned char`.

4. Results

This section presents preliminary test results on interoperability, legacy code integration, scalability, and performance of clients and services developed with *gSOAP*.

4.1. Test 1: Interoperability

gSOAP participates in White Mesa's interop lab [22], which is the premier site for SOAP toolkit development and interop testing. In addition, example clients and services were developed with *gSOAP* to test interoperability with various real-world services offered by Xmethods [23], see Table 1.

Name	*Owner*	*Toolkit*
Delayed Stock Quotes	XMethods	GLUE
Currency Exchange	XMethods	GLUE
XMethods Filesystem	XMethods	Apache
XMethods Listings	XMethods	Apache
Flight Tracker	ObjectSpace	Apache
Who Is	Shiv Kumar	Delphi
Calculator	XML Components	XMLCLX
UDDI Proxy Service	DSTC Pty Ltd	MS. NET
Glossary	Luhala	SOAPLite

Table 1. Services Used in Interop Tests

```
n1__getQuote(char *symbol, float &result);
n2__getRate(char *country1, char *country2,
                                          float &result);
n3__getQuote(char *symbol, char *country, float &result);
```

Figure 6. `quotex.h`

```
main()
{ soap_serve(); // wait for request and call skeleton
}
char endpt[] = "http://services.xmethods.net/soap";
n3__getQuote(char *symbol, char *country, float &result)
{ float q, r;
  if (soap_call_n1__getQuote(endpt, "", symbol, q)
   || soap_call_n2__getRate(endpt, "", "us", country, r))
    return SOAP_FAULT; // pass exception on to the caller
  result = q*r;
  return SOAP_OK; // all OK
}
```

Figure 7. `quotex.cpp`

We illustrate how the functionalities of the XMethods Delayed Stock Quote and Currency Exchange services can be combined into one new Web Service that accepts a stock ticker name and a currency symbol and returns the currency-converted stock quote. This new service acts both as a server and as a client: after accepting stock ticker name and currency symbol from a client it communicates with the XMethods services to retrieve the stock quote in a dollar amount and to convert the quote into the requested currency.

Figure 6 shows the header file input to `soapcpp` that serves as the interface definition of the three remote methods of the three Web Services involved. Each remote method is specified as a function prototype. The remote method name, parameter names, and types are specific to each service and can be extracted from the published WSDL descriptions. Because C lacks a means for explicitly denoting in/out parameter passing modes, `soapcpp` uses the convention that the last parameter of the function is the output parameter. The integer return value is used for SOAP exception handling and indicates success or failure.

Figure 7 depicts the full server code (except for the namespace URI table which is omitted from the figure). When the service is deployed as a simple CGI application, the `soap_serve` routine called in `main` waits for requests on the input stream, inspects the intended remote method invocation from the XML payload, and calls the appropriate skeleton routine. The skeleton routine demarshalls the request and calls the `n3__getQuote` routine. This routine in turn calls the proxies of the `n1__getQuote` and `n2__getRate` remote methods of the XMethods services to retrieve the current stock quote and currency exchange rate. The routine computes the currency-converted stock quote and returns `SOAP_OK` indicating success. The skeleton converts the result in a SOAP compliant response and sends the response back to the client.

```
class ivector
{ int *__ptr; int __size; ... // class methods };
class vector
{ double *__ptr; int __size; ... // class methods };
class matrix
{ vector *__ptr; int __size; ... // class methods };
ns__ludcmp(matrix *a, struct ns__ludcmpResponse
               {matrix a; ivector i; double d;} &result);
```

Figure 8. `lu.h`

```
ns__ludcmp(matrix *a, struct ns__ludcmpResponse &result)
{ result.a = *a;
  return ludcmp(result.a, result.i, result.d);
}
ludcmp(matrix &a, ivector &i, double &d) { ... }
```

Figure 9. `luserver.cpp`

4.2. Test 2: Legacy Code Integration

The *gSOAP* stub and skeleton compiler generates marshalling routines for user-defined C/C++ data structures, thereby enabling the integration of legacy codes in SOAP applications. Consider for example the LU decomposition algorithm `ludcmp` and related routines `lubksb` (backsubstitution), `lusol` (solver), and `luinv` (inversion) of Numerical Recipies in C [6]. We implemented these in a Web Service with *gSOAP*. Figure 8 depicts the header file segment to declare `ludcmp`. The `ns__ludcmpResponse` struct contains the output parameters: the decomposed matrix `a`, reordering index vector `i`, and a double `d`.

Figure 9 depicts the server code with the implementation of the `ns__ludcmp` remote method. The algorithms in Numerical Recipies in C use dynamic array structures for matrices and vectors, i.e. a matrix is an array of pointers to arrays. We implemented these as `matrix`, `vector`, and `ivector` classes with pointers and size information. The size information is required for serialization and deserialization (see also Section 3.3). The use of classes for arrays did not require significant changes to the original `ludcmp` routine. The `ns__ludcmp` function is a simple interface to the `ludcmp` routine. Because `ludcmp` applies *in situ* LU decomposition, prior to calling `ludcmp` the interface sets the output matrix reference to the input matrix pointer.

Figure 10 illustrates the service architecture. The SOAP interface for the service is generated by `soapcpp`. The in-

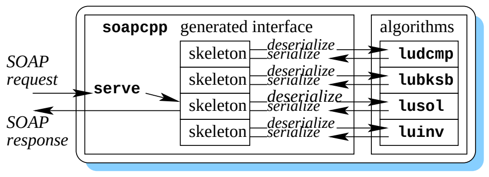

Figure 10. Linear Solver Service

terface consists of the `soap_serve` routine that dispatches the requests to the generated skeleton routines of the remote methods. The skeleton demarshalls the request parameters, calls the solver routine, (e.g. `lusol` in the figure), marshalls the result parameters, and sends the response back to the client. The linear solver service runs as a CGI application or as a stand-alone server serving requests via BSD Unix sockets. `soapcpp` also generates a WSDL description for service discovery and client access.

This example demonstrates that minimal effort is required to incorporate legacy C routines in a Web Service. More importantly, the critical data structures of the routines (pointers to arrays of doubles) could be used, however with the necessary embedding of array size information for the *gSOAP* compiler-generated (de)serializers.

4.3. Test 3: Scalability and Performance

The payload of a SOAP remote method request and response message requires more bandwidth than protocols that adopt binary serialization formats such as Java's object serialization format. To improve efficiency, a multi-protocol approach can be used [9] but that complicates implementation by requiring multiple protocol APIs. Instead, compression methods such as HTTP 1.1 gzipped transfer encodings should be used with SOAP to significantly reduce bandwidth demands[2].

The *gSOAP* stub and skeleton compiler improves marshalling performance significantly by providing fast *in situ* (de)serialization of data structures. SOAP C/C++ client and service implementations tend to be much more efficient compared to implementations in other languages. We observed that Perl SOAP::Lite and Apache 2.2 SOAP client implementations of the XMethods Delayed Stock Quote service were respectively about 3 times slower and 2 times slower compared to a *gSOAP* client implementation in C.

We tested the scalability and performance of the LU decomposition Web Service and a Web Service that we developed to produce magic squares. Both services have a high bandwidth demand because of the matrices and vectors contained in the request and response messages. The service applications were compiled with gcc and tested on a dual Pentium III 550MHz machine with 1G memory and 256K level-1 cache running Red Hat Linux. The client applications were compiled with gcc and tested on a dual Pentium III 933MHz machine with 256K memory and 256K level-1 cache running Red Hat Linux. The machines are connected with a 10BaseT Ethernet LAN with low to moderate traffic. Tests were performed by invoking remote methods a 100 times and during different times a day.

On the dual Pentium III 550MHz machine, the marshalling routines achieved ≈60,000 two-stage "count and send" serializations[3] per second and the demarshalling routines achieved ≈90,000 deserializations per second over the standard 10BaseT Ethernet LAN.

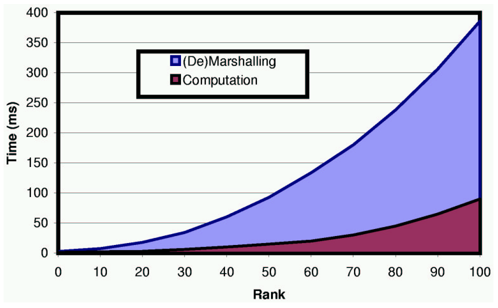

Figure 11. Elapsed Time (ms) of Remote Method Invocation of `luinv` by a Client

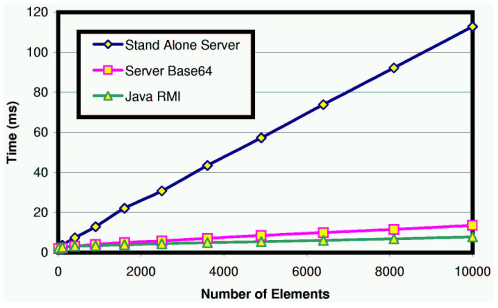

Figure 12. Elapsed Time (ms) of Remote Method Invocation by a Client of a Magic Square Service Compared to Java RMI

Figure 11 shows the (de)marshalling+communication overhead compared to the compute time for matrices up to order 100. The (de)marshalling+communication overhead is the total time for marshalling a matrix by the client stub, sending the uncompressed matrix over the 10BaseT Ethernet, demarshalling the matrix by the skeleton, marshalling the inverted matrix, and sending the inverted matrix back to the client who demarshalls it. The compute time for matrix inversion by `luinv` accounts for ≈20% of the total elapsed time. The remaining 80% is largely consumed by the overhead introduced by (de)marshalling and communication.

Figure 12 shows the scalability and performance of SOAP remote method invocation by a client of the magic squares service compared to a Java 1.2.2 RMI Linux implementation of a magic squares service and client (the matrix is a Java int[][] array). Two SOAP service implementations were tested: one with the 32-bit integer matrices marshalled as SOAP arrays and one that uses SOAP Base64 encoding. A Base64 binary encoded matrix takes 33% more space compared to internal storage, but is far less demanding on

[2]XML compression rates are typically very high.

[3]One serialization of one double floating point value.

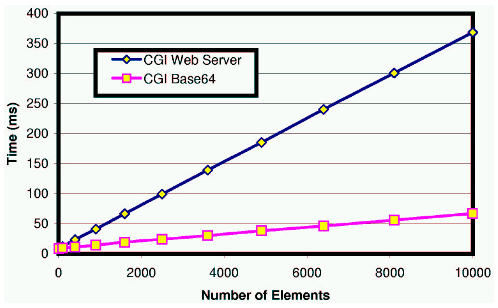

Figure 13. Elapsed Time (ms) of Remote Method Invocation by a Client of a CGI-Based Magic Square Web Service

bandwidth compared to SOAP arrays. The x-axis denotes the size of the matrix as the number of matrix elements to illustrate the linear scalability of the remote method invocation (the magic squares algorithm is linear in the number of elements of a matrix). Java RMI performance is better for larger matrices due to optimizations in the Java RMI socket layer which have not yet been implemented in *gSOAP*.

Figure 13 depicts the elapsed times of the two different magic squares service implementations deployed as CGI applications on an Apache Web server 1.3.22 running on a dual Pentium III 550MHz machine with 1G memory and 256K level-1 cache. The services are scalable, but CGI introduces a significant overhead.

5. Conclusions

The SOAP protocol and the abundance of SOAP toolkits promise to make the development of business and science portals with programming- and platform-neutral Web Services easy and very likely in the near future. The *gSOAP* stub and skeleton compiler provides a unique SOAP-to-C++ language binding for the development of SOAP-enabled applications such as clients, services, and peers. Other SOAP C++ toolkits adopt a SOAP-centric view and offer SOAP APIs for C++ that require the use of class libraries for SOAP-like data structures. This solution often forces a user to adapt the application logic to these libraries, which is undesirable for the integration of legacy applications, embedded, and real-time systems. In contrast, *gSOAP* provides transparent SOAP API through the use of compiler technology that hides irrelevant SOAP-specific details from the user. The compiler automatically maps native and user-defined C and C++ data types to semantically equivalent SOAP data types and vice-versa. As a result, full SOAP interoperability is achieved with a simple API relieving the user from the burden of SOAP details and enabling him or her to concentrate on the application-essential logic.

References

[1] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the web. In *The 9th Int'l Conference on Parallel and Distributed Computing Systems*, 1996.

[2] Beck et al. HARNESS: A next generation distributed virtual machine. *Future Generation Computer Systems*, 15, 1999.

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

[4] A. Brown et al. SOAP security extensions: Digital signature. Technical report, W3C, 2001. http://www.w3.org/TR/SOAP-dsig.

[5] T. Chilimbi, M. Hill, and J. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[6] B. Flannery, W. Press, S. Teukolsky, and W. Vettering. *Numerical Recipes in C*. Cambridge University Press, Cambridge UK, 2nd edition, 1992.

[7] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Jnl. of Supercomputer Applications*, 11(2):115–128, 1997.

[8] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid. *International Journal of Supercomputer Applications*, 15(3):200–222, 2001.

[9] M. Govindaraju et al. Requirements for and evaluation of RMI protocols for scientific computing. In *Supercomputing*, 2000.

[10] A. Grimshaw and W. Wulf. The legion vision of a worldwide virtual computer. *CACM*, 40(1):39–45, 1997.

[11] P. Kulchenko. SOAP::Lite for Perl. http://www.soaplite.com.

[12] J. Moreira, S. Midkiff, M. Gupta, P. Artigas, P. Wu, and G. Almasi. The NINJA project. *CACM*, 44(10):102–109, 2001.

[13] M. Neary, S. Brydon, P. Kmiec, S. Rollins, and P. Cappello. Javalin++: Scalability issues in global computing. In *ACM Java Grande*, pages 171–180, San Fransisco, CA, 1999.

[14] M. Neary, B. Christansen, P. Cappello, and K. Schauser. Javalin: Parallel computing on the internet. *Future Generation Computing Systems*, 15:659–674, 1999.

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

[16] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *ACM SIGPLAN Converence on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[17] Sun Microsystems. Enterprise java beans technology. http://java.sun.com/products/ejb/.

[18] Sun Microsystems. Project JXTA: Technical specification, version 1.0, 2001.

[19] UDDI. The universal description, discovery, and integration (UDDI) specification. http://www.uddi.org/specification.html.

[20] R. van Engelen. The gSOAP toolkit 2.1, 2001. http://sourceforge.net/projects/gsoap2.

[21] R. van Engelen, K. Gallivan, G. Gupta, and G. Cybenko. XML-RPC agents for distributed scientific computing. In *IMACS'2000 Conference*, Lausanne, Switzerland, 2000.

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

[23] XMethods. XMethods service listings. http://www.xmethods.com.