A Benchmark Suite for SOAP-based Communication in Grid Web Services

Michael R. Head[1], Madhusudhan Govindaraju[1], Aleksander Slominski[2], Pu Liu[1],
Nayef Abu-Ghazaleh[1], Robert van Engelen[3], Kenneth Chiu[1], Michael J. Lewis[1]

1. Grid Computing Research Laboratory (GCRL), State University of New York (SUNY) at Binghamton
2. Computer Science Department, Indiana University
3. Department of Computer Science, Florida State University

Abstract

The convergence of Web services and grid computing has promoted SOAP, a widely used Web services protocol, into a prominent protocol for a wide variety of grid applications. These applications differ widely in the characteristics of their respective SOAP messages, and also in their performance requirements. To make the right decisions, an application developer must thus understand the complex dependencies between the SOAP implementation and the application. We propose a standard benchmark suite for quantifying, comparing, and contrasting the performance of SOAP implementations under a wide range of representative use cases. The benchmarks are defined by a set of WSDL documents. To demonstrate the utility of the benchmarks and to provide a snapshot of the current SOAP implementation landscape, we report the performance of many different SOAP implementations (gSOAP, AxisJava, XSUL and bSOAP) on the benchmarks, and draw conclusions about their current performance characteristics. [1]

Key Words*: SOAP, Web Services, Grid Communication, Benchmark.* [2]

1 Introduction

Web services have emerged as the architecture of choice for grid standards such as the Web Services Resource Framework (WSRF) [14]. The WSRF initiative represents a re-engineering of grid computing standards to be compatible with the current Web services conventions and specifications. This convergence simplifies the design, development and deployment of grid services in virtual organizations with diverse compute and resource characteristics.

Two important Web services specifications are Web Services Description Language (WSDL) [11] and SOAP [17]. WSDL provides a standard language to precisely specify all the information necessary for communication with a Web service, including the interface of the service, its location, and the list of communication protocols it supports.

SOAP is the most commonly used Web services communication protocol for information exchange in a distributed and heterogeneous environment. SOAP specifies how a message—and the data within it—may be represented and wrapped in XML. Though the SOAP specification permits the use of any transport protocol, HTTP is the de facto standard due to its widespread use.

The mapping of method signatures into request-response formats, required by an RPC framework, is achieved via conventions stated in the specification. SOAP supports one-way messaging and various request-response type exchanges including RPC.

The combination of XML and HTTP lends many attractive traits to SOAP, including simplicity, expressiveness, platform and language independence, extensibility and robustness. SOAP is a popular choice as the common underlying protocol for interoperability between grid services. These features facilitate the use of SOAP in diverse applications with widely varying characteristics and requirements. Clients and grid Web service endpoints can also add optimizations in their implementations, without making limiting assumptions about the capabilities and configuration of potential receivers of these messages. Though SOAP has many vital features to offer to grid applications, the specification has not been developed with performance as an important goal.

This has spurred our research towards the study of the bottlenecks that are inherent to the protocols, and the development of novel optimizations to attain the best possible perfor-

[1]Supported in part by NSF grants: ANI-0330613, CCR-0208892, IIS-0414981, CNS-0454298, Career Award ACI-0133838, and DOE grants DEFC02-01ER25451, DEFG02-02ER25526, Early Career Principle Investigator grant DEFG02-02ER25543.

SC—05 November 12-18, 2005, Seattle, Washington, USA (c) 2005 ACM 1-59593-061-2/05 /0011...$5.00

mance within the constraints of the specification.

The convergence of grid and Web services standards has elevated the importance of SOAP, requiring the evaluation of SOAP for data types and communication patterns used by grid applications. It is thus important to have a test framework to determine if a particular SOAP toolkit can meet the performance requirements of an application, or if some other communication protocol should be employed. SOAP implementations are interesting and important to compare, contrast, and evaluate so users can make informed decisions based on evidence of claimed benefits. A comprehensive benchmark suite is necessary because:

- The requirements on any communication substrate for grid services varies for different applications. These requirements could include low latency, high throughput communication, minimal memory footprint, optimizations to handle scientific data, efficient handling of large message sizes, capabilities to compress messages on the fly, and support for streaming capabilities. Such disparate requirements lead to a wide range of possible design and implementation choices. The benchmark suite can aid in determining the toolkit that has the most optimized SOAP implementation for a specific feature of interest.
- A simple and straight forward implementation of various SOAP modules can limit performance. A good benchmark can help developers identify where optimization may be useful. Ideally, SOAP toolkits should dynamically switch to clever optimizations for specific data structures, use cases and communication patterns. We discuss many such techniques in Section 2.
- Grid users have a wide range of SOAP implementation choices [26]. SOAP toolkits exist in languages such as Java, C, C++, Perl, Python and C#. It is important to have a standard mechanism to quantify, compare, and evaluate the performance of each toolkit and study the strengths and weaknesses for a wide range of representative use case scenarios.

For these reasons, we have designed and developed a common standard SOAP benchmark suite for testing the performance and scalability of different SOAP toolkits, with a focus on data structures commonly used in grid services. In designing the benchmarks, we draw on our experience in implementing and optimizing the performance of SOAP's various features for three different independent SOAP toolkits, gSOAP [30, 31, 32], XSUL [16, 20, 25], and bSOAP [4, 5, 6].

One contribution of this paper is in the design and specification of the benchmark. This suite provides SOAP implementors with working examples of SOAP features, and gives them a way of testing and assessing the performance of their specific implementation of these features. Another contribution is the snapshot it provides of the current performance of many popular SOAP implementations. This performance study provides insight into the relative strengths and weaknesses of different SOAP implementations under different usage scenarios, and demonstrates the utility of the benchmark suite. The benchmark suite and driver programs can be used to continuously compare the performance of available toolkits. The performance results in this paper show how effectively the benchmark suite can be used to select an appropriate SOAP toolkit for specific application needs.

Our work will benefit both SOAP developers and grid applications programmers alike. SOAP library developers can gain insights into the various factors and design choices that determine the performance of a SOAP toolkit, thereby improving their ability to build better faster implementations. Application developers can use the benchmark suite to test and compare the performance of various aspects of different toolkits, and accordingly select the one that best suits their application's needs. We include both floating-point and base64 performance results so that scientific users can decide whether or not they will need to send their numerical data as base64-encoded binary. We ran our benchmarks on the following widely used toolkits: gSOAP, AxisJava, .NET, and XSUL. We also present results for bSOAP, which is an emerging SOAP toolkit.

The remainder of this paper is organized as follows. Section 2 provides the motivation, description and insights into the benchmark suite that we have designed. Section 3 describes our experimental setup and a representative set of performance results. We present a set of observations that can be drawn from our test results in Section 4. We discuss related work in Section 5 and end with pointers to future work in Section 6.

2 Benchmark Suite

The benchmark suite consists of operations in WSDL files along with bindings for SOAP calls and a driver that reads trace data from local files and automates the testing process. Each SOAP toolkit is made to implement the operations defined in the WSDL documents for the benchmark. This section explains the rationale for each benchmark's design, and describes various optimizations that can be used to improve the performance of a toolkit for the features exercised by the benchmark.

2.1 Serialization, Deserialization and Round-Trip Performance

SOAP has been expressly designed to support interoperability between multiple different implementations. It is important to collect and analyze isolated performance statistics for serialization and deserialization because the toolkits used by clients and servers may differ. With the recent release of WSRF implementations [15], we expect a wide range of

SOAP implementations to interact with well known WSRF services.

2.1.1 Serialization

SOAP serialization converts in-memory objects into an XML stream that is sent on the wire in UTF-8 format. We list several optimizations that address various stages of serialization.

HTTP 1.0 requires the precise buffer length to be placed in the HTTP header's *Content-Length* field. A simple SOAP serialization algorithm allocates and extends the buffer as the data structures are traversed and converted to ASCII. Once this conversion is completed, the buffer length is calculated and added to the HTTP header. This naive approach can invoke multiple expensive memory operations to create a single large memory buffer with the SOAP payload.

In [9] we describe how to avoid reading each character of the XML tags from memory. If the tags are created as literals, the characters comprising the tags are likely to be in the instruction stream as immediate operands.

bSOAP [6] reduces the number of system calls by using *vectored send* to dispatch multiple buffers with a single call. gSOAP [32], on the other hand, uses a two-iteration algorithm. The first iteration traverses the data structures and calculates the required buffer length. The second iteration generates the HTTP header, fills in the content length, and serializes the SOAP message directly over TCP/IP. This approach avoids keeping the entire buffer in memory.

In earlier work we showed that the conversion of IEEE 754 floating point data to ASCII is complex and can account for 90% of end-to-end communication time [9]. Our analysis with Sun Forte on a Blade 1000 determined a sharp drop in performance when the required precision ranges from 14 digits to 17 digits.

The serialization bottleneck for floating point data is addressed in bSOAP through *differential serialization* [6]. The idea behind differential serialization is to store copies of previously sent messages within the stub that sends them. When the stub is invoked for future calls, only the data that has been changed needs to be re-serialized into the message. The extent to which this optimization is effective depends on message size, content, structure, and the similarity between consecutive messages. Shifting message contents in memory, stealing space from neighboring fields, and stuffing fields with whitespace can increase the cases when differential serialization can be applied [5]. Our initial performance results demonstrate that differential serialization can improve send-times between 17 and 1000 percent [4, 5, 6].

Serialization Benchmark: Our benchmark suite measures the serialization performance of various toolkits for arrays of different data types and sizes that are often used in grid applications. The significance of memory management, conversion to ASCII formats, cost of establishing TCP connections, and size of cache varies as the size of the array changes. The benchmark driver generates a request to invoke a method on the SOAP toolkit being tested, which serializes an array of the requested size and type. The driver sends the invocation for several iterations and measures the average performance of the toolkit.

2.1.2 Deserialization

Deserialization converts XML streams in wire-format to objects in memory. We discuss several important aspects of the deserialization process.

The widely used paradigms for parsing XML documents include Document Object Model (DOM), Simple API for XML (SAX) and XML Pull Parser (XPP) [20].

The DOM model maps the XML document into a tree representation in memory. This allows the document to be easily traversed and modified. However, for large documents, DOM parsing can be memory intensive. In contrast, SAX parsing never stores the entire XML document in memory. Instead, a callback model emits events for all the document's elements and tags. For large static documents, SAX is preferable to DOM. SAX is also often used when only a few specific elements of an XML document need to be extracted and processed. *Pull parsing*, employed by the XPP parser, is specialized for parsing payloads in which elements are processed in succession, and no element needs to be revisited. XPP provides the added feature of building a partial XML Infoset tree in memory in an incremental manner.

DOM, SAX, and XPP require two passes through the XML document; the parser tokenizes the document in the first pass, and the application processes the content in the second. An STL `map` is typically used to compare each tag retrieved by the parser with the one that is expected. Results in [9] show that a `trie` data structure, which has $O(1)$ lookups as opposed to $O(\lg n)$ for STL `map`, can provide significant performance improvement for matching tags that appear repeatedly.

gSOAP uses a performance-aware compiler that generates code for fast XML parsing and processing of native C/C++ types. The algorithms are based on single-pass schema-specific recursive-descent parsers for XML decoding and dual pass encoding of the application's object graphs in XML.

Deserialization Benchmark: Our benchmark consists of SOAP messages for different sizes of frequently used data types (strings, integers and doubles). Payload elements are fully namespace qualified and the driver verifies that the toolkit has appropriately handled all elements. Deserialization benchmarks for complex types are described later. Our tests include toolkits that use different models for parsing XML documents: XSUL uses XPP, gSOAP uses a recursive descent parser, AxisJava can use SAX or DOM, and .NET uses an efficient streaming parser via the `XMLReader` class. The benchmark driver sends trace data with SOAP payloads of various sizes (and for various types of data) and invokes the method on the SOAP toolkit requiring it to deserialize the request. The driver repeats this test for several iterations for each toolkit and measures the performance.

2.1.3 End-to-End Performance

The determining factors for end-to-end performance are serialization, deserialization, and available network bandwidth. If the same toolkit is used for both the client and Web service implementation, deserialization could be optimized for the parameters set by the serialization process. Even though the wire protocol is fixed for SOAP (1.1), the names of the tags, placement of whitespaces, and serialization order of data members of an object are not fixed. Apart from SOAP payload specific parameters, the TCP packet size and the size of each chunk can be fine tuned if the same toolkit is used by a sender-receiver pair.

End-to-End Performance Benchmark: This benchmark combines the tests for serialization and deserialization. The driver measures toolkit scalability for all primitive data types, and for arrays of primitives. A different benchmark measures complex data types, discussed later. This benchmark's example users include those who download WSRF [15] and use the default implementation, based on AxisJava, for both the client and service endpoints.

Results in [23] show that Base64 parameter encoding can significantly reduce overhead, compared to standard XML encoding. To quantify this advantage, for all types and sizes, we have included it in the benchmark suite for serialization, deserialization, and end-to-end performance measurement. The end-to-end benchmark driver sends trace data for serialized SOAP messages (of various types and sizes) to the SOAP toolkit being tested. The toolkit is required to deserialize this message and again serialize it as the return value for the method invocation. The driver repeats this process over several iterations for each data type and size.

2.2 Candidate Features for Optimizations

In this section we isolate the optimizations and related benchmarks for specific SOAP implementation features. The measurement methodology for the benchmark driver is the same as described in the subsection above.

2.2.1 Streaming vs Non-Streaming

As discussed earlier, calculation of HTTP 1.0 content length of the SOAP payload can hinder serialization performance. We showed in [16] that the size of SOAP's on-the-wire representation is a factor of four to ten times greater than the corresponding binary representation. SOAP processing is affected by both extra memory invocation for calculating the length, and cache misses due to large buffers. HTTP 1.1's chunking and streaming feature can help address this problem.

HTTP 1.1 explicitly supports overlapping the serialization process with the network transmission of buffers [31]. *Persistent connections* (keep-alive) reuse the same TCP/IP connection for multiple calls between two endpoints. The SOAP payload can be sent in chunks, with each chunk preceded by its length. Small chunk sizes can ensure cache hits but result in many system calls. Large chunk sizes can reduce the number of system calls, but may not always lead to cache hits. Thus, chunk size should be a configurable parameter, set according to the native system characteristics.

Streaming Benchmark: The streaming benchmark measures the performance of serialization, deserialization, and end-to-end performance when chunking and streaming is used with a persistent TCP/IP connection, compared to the case when it is turned off. For small messages, the cost of repeatedly establishing network connections can significantly impact performance. This benchmark quantifies the exact performance benefit of using streaming. The driver sends streaming data from trace data for primitives, arrays and complex types. It can be configured to vary the size of each chunk in the SOAP payload.

2.2.2 Namespaces

XML namespaces are extensively used in SOAP messages. Namespaces are used to uniquely identify and distinguish between identical names of tags, elements and attributes. Each namespace is associated with a defining namespace name (URI). Each tag has a prefix that points to a fully qualified name via a special attribute named *xmlns*. XML parsers typically use a stack to store namespace prefixes and corresponding URIs. The number of defining *xmlns* namespace bindings in an XML message is typically much smaller than the number of uses of this namespace prefix. As a result, maintaining the stack can result in several comparison operations, and can hurt the overall performance of the deserialization module.

Processing of *xmlns* attributes can be optimized by using just one table lookup to determine a corresponding internal namespace prefix. The table should be populated with prefixes obtained from the XML schemas of the SOAP messages. The namespace stack can simply record the translated prefixes to provide efficient matching of qualified tags and avoid storage and expensive comparison of namespace URIs.

Namespace benchmark: Our benchmark consists of SOAP payloads with varying level of nested data structures (linked lists). Several of the tag names and attribute names in each level are identical, forcing toolkits to correctly resolve them according to the namespace qualifications. The payloads also have a varying number of namespace qualified attributes. The synthetic data for these benchmark is based on the various attributes and nested elements required for emerging security standards for grid Web services, such as WS-Security [18]. The WS-Security standard enhances the SOAP messaging protocol to provide message integrity and confidentiality by associating tokens (included as namespace qualified elements and attributes) with each message.

2.2.3 Multi-ref

Each co-referenced object is assigned a unique identifier, represented by an attribute value, when it is serialized the first time. If the same object appears again in a data structure, it can be serialized with a multi-ref accessor, *id-ref*, that points to the identifier of the original object. Multi-ref is essential to efficiently serialize cyclic data-structures. This design is analogous to the use of pointers and references in many programming languages to refer to one instance of an object from multiple locations. The serialization of co-referenced objects require serialized objects to be stored in a table. Before an object can be serialized, the table needs to be searched to determine if the object has already been serialized, in which case the *id-ref* attribute must be used. A naive implementation can hurt the scalability of the serialization process. For example, if the *equals()* method has to be invoked on each object, as is often done in Java, serialization of *n* objects will lead to an $O(n^2)$ serialization algorithm. For toolkits based in Java, we recommend the use of *IdentityHashMap*, which is optimized for use with with Java references.

Multi-ref Benchmark Our benchmark consists of an array of strings, wherein many of the strings are identical. A multi-ref compliant toolkit must check for co-references for every string. This is usually done via lookups in a hash-table. However, due to hash table's overflow chains, it may not always perform a lookup in constant time. As the array size increases, the overhead of maintaining the logical coherence of graph structures negatively affects performance.

2.3 Latency

We define *latency* as the overhead incurred by a toolkit in an end-to-end call when no parameters are sent or received. It quantifies the minimum response time of a toolkit. However, the latency measurement does not include costs associated with *cold start* or *warmup*, such as initialization costs due to loading of the necessary dynamic libraries or Java class files. Measurements are taken after the first few iterations.

Latency Benchmark: The benchmark is `void echoVoid()` operations to test the *overhead* imposed by a toolkit. Even though no parameters are sent, the call still traverses all the layers of the serialization and deserialization stack, and effectively measures the overhead that will be inherent to every call.

2.3.1 Application Specific Benchmarks

We describe some benchmarks that are based on well known services and widely used communication patterns used in distributed applications.

2.3.2 Events

An event can be broadly defined as a time-stamped message with typed data that is delivered from a source to a set of subscribed listeners. Events provide a de-coupled communication medium for grid applications. Typical uses of events include monitoring, debugging, and reporting occurrences such as a successful creation of a remote file. Event services (also called *notification* services) should be extensible, language-independent, platform independent and provide ready integration with applications. The SOAP protocol is perfectly suited to meet these requirements. Notification services were recognized as a common port type in the OGSI [12] specification.

Events Benchmark: We have defined the event data structure as a complex type with three data members: an integer (sequence number), a double (time stamp) and a string to store the event message. This definition provides both simplicity and flexibility. The string can be used to store small values such as a *url* for GridFTP transfer, or a long string requesting resource properties from a WSRF service.

Our benchmark driver measures the performance of a toolkit for sending and receiving events ranging from tens to thousands of events. The driver can be configured to choose a string size that accurately reflects the needs of events in the application of interest.

2.3.3 Mesh Interface Objects

Scientific components on the grid frequently exchange *mesh interface objects* (MIO) structures of the form (int, int, double). The two integers represent a mesh coordinate and the double represents the field value. A typical use of MIOs is in communication between two partial differential equation (PDE) solvers on different domains. Example applications include a climate model that ties together an atmospheric simulator with an ocean circulation simulator [10] and fluid simulation that is coupled with a solids structure code [19].

MIO Benchmark: The performance tests for MIOs record the scalability of a SOAP toolkit as the number of MIOs is varied from ten to 25,000.

2.3.4 MCS Benchmark

The Metadata Catalog Service (MCS) [24] is a well known grid service that provides a framework for efficiently managing the storage and retrieval of metadata associated with large collections of files generated by data-intensive applications. Clients of MCS interact with the MCS Web service via the Axis [28] SOAP implementation. A scalability study in [24] shows that the Web service overhead causes an average performance drop by a factor of 4.8. We contacted the authors of [24] to obtain the synthetic data (for the names, types and values of the attributes) used for their tests, and used it to define this benchmark. Each attribute is a tuple consisting of a name, type, and value. Our tests can be used to determine which is the best available toolkit for MCS.

2.3.5 Google Web Service

The Web service interface for the Google search engine [2] and Amazon.com [1] portal, are the two most widely used industrial Web services. Though they are not grid services, they are representative of the growing interest in using Web services based protocols. Both Google and Amazon.com sites receive a large volume of requests daily, and serving these requests via SOAP can be expensive. Our benchmark measures the performance of processing the response from the `doGoogleSearch` query of the Google Web service API.

2.4 Binary Attachments via DIME and MIME

Two important protocols for sending binary large objects (BLOBs) as attachments are Direct Internet Message Encapsulation (DIME) [33] and Multipurpose Internet Mail Extensions (MIME) via SOAP with Attachments (SwA) specification [33]. The motivation for sending binary data as attachments is to avoid the overhead of serialization and deserialization for binary data. Moreover, for digitally signed attachments, it may not be possible to use standard serialization techniques without affecting the integrity of the data. MIME messages are sent as a series of records, with each record separated from the other via a unique marker string. DIME and MIME have many similarities. The important difference is that in DIME, the header for each record contains the exact size of the record. If multiple attachments have been sent, the receiver can directly access a particular record.

Our benchmark sends multiple attachments, one to fifty, varying the size of the attachment from 1KB to 100KB. The aim of the benchmark is to determine the threshold point when it is better to use DIME instead of MIME (or vice versa).

2.5 Dynamic vs Static Code Generation

The Java Dynamic Proxy Class [27], introduced in Java 1.3, is an elegant and flexible feature that allows a class to implement a list of interfaces specified at runtime. This design is in stark contrast to the use of classic `stubs` and `skeletons`. Stubs and skeletons shield run-time specific details from a user, and are generated by a specialized code generator. The dynamic proxy feature obviates the need for a code generator in Java based SOAP toolkits, and provides a type-safe reflective dispatch of invocation on dynamically created proxies. The dynamic proxy, however, imposes a severe performance penalty. Even though the generation of static stubs and skeletons for every server interface is cumbersome, it offers attractive performance benefits. Our benchmark is designed to illustrate the exact performance penalty of using dynamic proxies. For this test, the toolkit must provide support for switching between the two designs.

Linux toolkit	**gSOAP**	**XSUL**	**AxisJava**
Latency (seconds)	0.00052	0.00107	0.01200
Windows toolkit	**.NET**	**XSUL**	**AxisJava**
Latency (seconds)	0.0035	0.0038	0.0047

Table 1. *Latency is measured by the time in seconds for the client to receive the full response from an* `echoVoid` *call. On Linux, XSUL's overhead is two times greater than gSOAP's. However, AxisJava's overhead is 10 times greater than XSUL's. On Windows, .NET and XSUL have comparable overheads, while AxisJava's overhead is 33% higher than .NET's.*

3 Performance

This section describes the performance of a representative set of five toolkits when run against our benchmarks. We used two sets of drivers. A heavy-weight driver verifies the accuracy of every response from the tested toolkits. A light-weight driver reads trace data from files into a local buffer, then sends it to the different toolkits for several iterations, but the responses are not checked for accuracy. This approach ensures toolkit accuracy, but keeps verification cost out of the reported performance data. The light-weight driver was configured to run each benchmark for multiple iterations and calculate the average time taken for each iteration.

The versions of the tested toolkits are: gSOAP 2.7e, AxisJava 1.2RC2, .NET 1.1.4322, XSUL 1.99RC2 and bSOAP 0.5alpha. gSOAP is implemented in C/C++, bSOAP in C++, and XSUL and AxisJava are developed in Java. Some performance variations are due to the inherent efficiency of C and C++ implementations over Java. However, since the SOAP wire-protocol is independent of any programming language and interoperability is an important feature of SOAP, comparing implementations in different languages is useful.

Since the .NET implementation is available only on a Windows platform, we compared its performance with the Java implementations of Axis and XSUL deployed on a Windows box. We also deployed gSOAP, AxisJava, XSUL and bSOAP on Linux machines and tested the performance separately. The results on Windows and Linux need to be viewed separately, as the hardware configurations of the Windows and Linux machines were not identical.

The Linux test environment consisted of two dual processor machines, each configured with 2.0 GHz Pentium 4 Xeon with 1GB DDR Ram and a 15K RPM 18GB Ultra-160 SCSI drive running Debian Linux 3.1 ("sarge") with the 2.4.26 kernel. The machines were connected by Gigabit Ethernet. gSOAP and bSOAP were compiled with gcc/g++ version 3.3.4. XSUL and AxisJava were compiled with Java 1.4.2. Relevant socket options, for both gSOAP and bSOAP, include SO_KEEPALIVE, TCP_NODELAY, SO_SNDBUF = 32768,

and SO_RCVBUF = 32768. For fairness, when bSOAP is compared with the other toolkits, it is set to re-serialize all data (BSOAP_100 means bSOAP is serializing 100% of its data).

For the tests on Windows (.NET, XSUL and AxisJava) we used a Dell Dimension 4500 with Intel Pentium 4 2.26GHz processor, 1GB of DDR SDRAM and 80GB Ultra ATA/100 hard drive running Windows XP.

The performance graphs show the measured time in seconds on the y-axis and the size of the application level data on the x-axis. The effective throughput (bandwidth obtained by a given SOAP implementation) can be calculated directly from the data points on each plot.

3.1 Summary of Performance Results

- **Latency**: Table 1 shows the overhead imposed by each toolkit, for both Windows and Linux platforms. gSOAP's overhead is less than XSUL by a factor of 2. XSUL outperforms AxisJava by a factor of ten. On Windows, AxisJava is also slower than .NET and XSUL. The overhead of gSOAP is lower than that of the Java-based toolkits (XSUL and AxisJava) as it does not incur the cost of using reflection and dynamic proxy classes. gSOAP uses statically generated stubs and skeletons for each remote call, which are known to be faster than dynamically generated code (proxies).
- **Serialization**: Figures 1 and 2 compare the serialization performance of doubles and integers respectively. XSUL and AxisJava perform similarly for all array sizes. The cost of the toolkit overhead is higher for AxisJava (as can be seen in the results for latency), but it gets amortized with increase in the size of data being sent. The cost of serialization for large array sizes (especially for floating point data) is dominated by the conversion of data from floating point representation to the ASCII format [9]. AxisJava and XSUL use the same conversion routines, and hence have similar performance characteristics for serialization.

  bSOAP is more efficient than gSOAP for all array sizes. For 25,000 elements bSOAP and gSOAP take 4% and 17% respectively, of the time it takes AxisJava to complete the benchmark. gSOAP uses a two-iteration serialization algorithm (described in 2.1.1), while bSOAP directly allocates buffers during the serialization process.
- **Deserialization**: We ran the deserialization benchmark for doubles and strings. The tested version of bSOAP did not have support for deserialization, and hence bSOAP is not included in the graphs. Figures 3 and 4 show that AxisJava does not scale well. For 10,000 elements, AxisJava takes 5 times more time than XSUL, and 35 times more than gSOAP to execute the benchmark. On Windows (see Figure 5), XSUL and .NET have similar performance. AxisJava's execution time exceeds .NET by a factor of 6. The choice of an XML parser plays a significant role in deserialization of SOAP payloads. XSUL uses an efficient pull parser (XPP) that has a low memory footprint and is specifically designed to access elements in a SOAP payload. AxisJava uses Xerces, which is modular and flexible but inefficient for large payloads. gSOAP also uses a custom pull parser during the deserialization phase.
- **Events and MIOs**: XSUL dynamically re-allocates memory as it retrieves new XML nodes while deserializing the XML graphs. This hurts its performance for events and MIOs (Figures 6 and 7). XSUL's performance exceeds AxisJava for less than 10,000 MIOs and 15,000 events. gSOAP outperforms both AxisJava and XSUL for all sizes of MIOs and events. Figure 8 shows a similar pattern for Windows-based toolkits; .NET outperforms AxisJava by 50% for large array sizes.
- **Base64 Encoding**: Figures 9 and 10 compare end-to-end and serialization performance respectively for Base64 encoding. Results show that gSOAP and XSUL outperform AxisJava. For end-to-end performance, AxisJava is slower than XSUL by a factor of 2.3, while XSUL is slower than gSOAP by a factor of 1.6. For serialization, XSUL takes 49% more time than gSOAP to complete the benchmark for 25,000 array elements.
- **End-to-End Performance**: When compared in isolation, XSUL outperforms AxisJava for deserialization but has similar performance for serialization. However, when the two modules are combined in Figures 11 and 12, XSUL outperforms AxisJava.
- **Chunking and Streaming**: We study the effect of chunking and streaming for deserialization of events (Figure 13) and serialization of doubles (Figure 14). For deserialization, the performance improvement is 22%. However, for serialization, AxisJava has no performance improvement, suggesting an inefficient buffering algorithm. gSOAP gains up to 42% with streaming for serialization of doubles.
- **Differential Serialization**: Figure 15 shows the performance improvement in bSOAP when differential serialization is used with different percentage of values changed from the previous run. The best case, when all the values are the same, is 6.7 times faster than the worst case, when all values need to be re-serialized. bSOAP serializes only those array elements that have changed since the previous send. So, for test cases where subsequent sends are similar, the performance of other toolkits will not change, while that of bSOAP will improve.

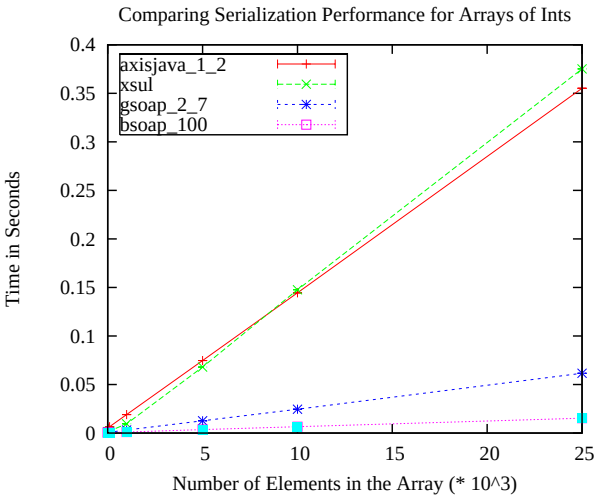

Figure 1. *This graph shows the effect on serialization when the size of an integer array is scaled. The performance of AxisJava and XSUL is similar. bSOAP and gSOAP complete the benchmark in 4% and 17% respectively of the time it takes for AxisJava for the largest size (25,000 integers).*

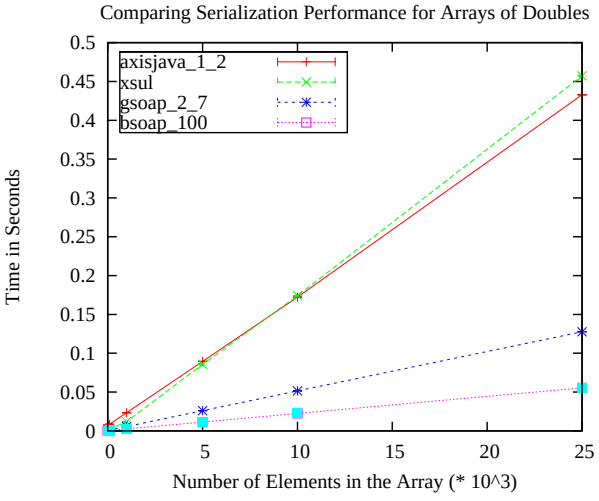

Figure 2. *Here we compare the serialization performance of four toolkits on arrays of doubles. As with the integer case in Figure 1, the Java-based toolkits (AxisJava and XSUL) perform similarly. bSOAP and gSOAP send the largest size (25,000 doubles) arrays in 13% and 30%, respectively, of the time it takes for AxisJava to do the same.*

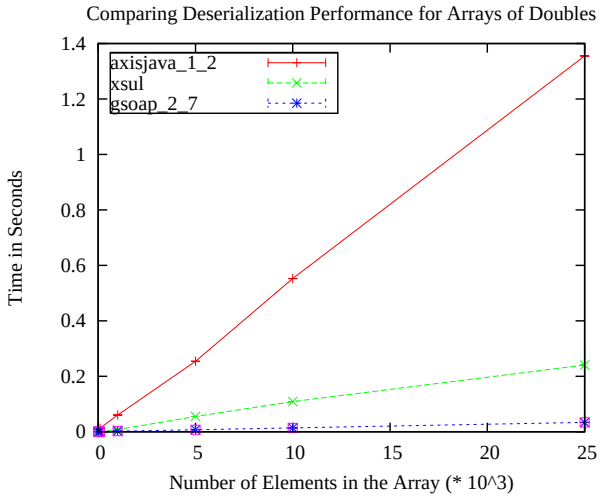

Figure 3. *Here we compare the deserialization performance of AxisJava, gSOAP and XSUL. Each toolkit is sent a SOAP payload for double arrays of various sizes, asked to deserialize it and return it's size to the driver. For arrays of 10,000 doubles, AxisJava takes 5.1 times as long to respond as XSUL, which in turn takes 7.8 times as long to respond as gSOAP.*

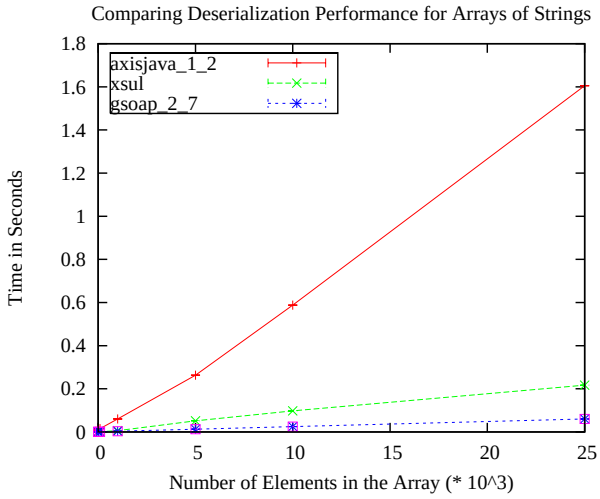

Figure 4. *This graph compares the deserialization performance for strings. Again, XSUL performs significantly better than AxisJava for deserialization. For an array size of 25,000, it takes AxisJava 7.4 times longer to respond than XSUL, and XSUL takes 3.6 times as long to respond as gSOAP.*

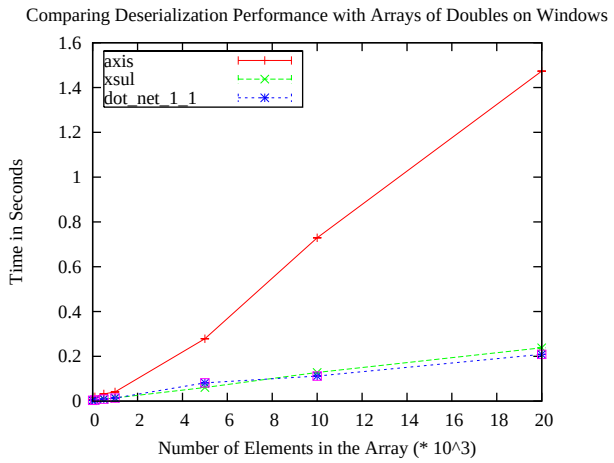

Figure 5. *This graph compares deserialization performance of Axis, XSUL and .NET for an array of doubles on Windows. XSUL and .NET are comparable, while AxisJava does not scale well for large array sizes. For an array of 10,000 doubles, AxisJava's deserialization time is 6 times greater than those of XSUL and .NET.*

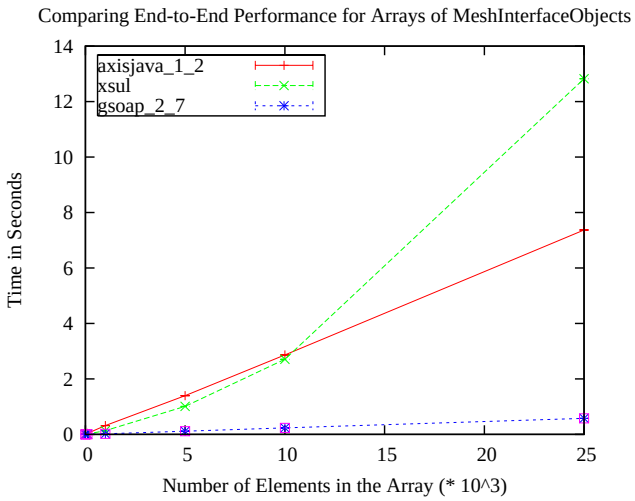

Figure 6. *We compare the end-to-end performance for arrays of MeshInterface Objects (2 integers and a double). The plots show that for sizes greater than 10,000 objects, XSUL's performance considerably degrades compared to AxisJava. For 10,000 elements, XSUL and AxisJava respectively take 11.7 and 12.3 times more time to execute the benchmark compared to gSOAP.*

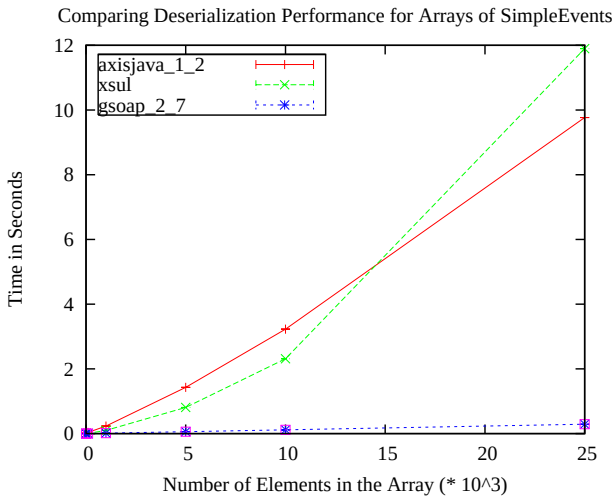

Figure 7. *This graph compares the performance for the deserialization benchmark for events. Each event object contains an integer, a double and a string. XSUL's performance degrades considerably when it deserializes more than 15,000 elements, but for lesser number of events, it outperforms AxisJava. gSOAP is orders of magnitude faster in handling complex types compared to the Java toolkits.*

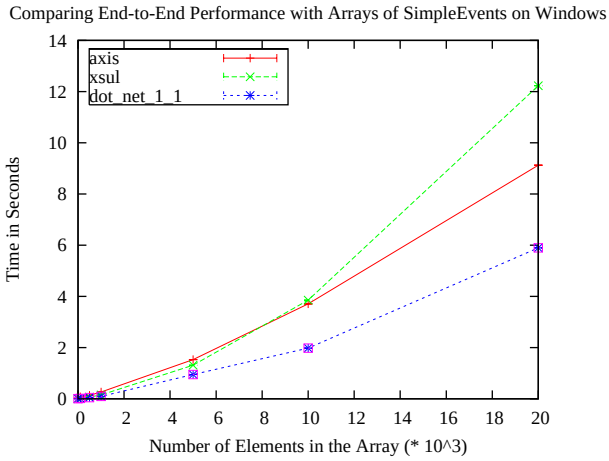

Figure 8. *We compare end-to-end (serialization and deserialization) for .NET, AxisJava, and XSUL on Windows. XSUL's performance drops for handling more than 10,000 events. For sizes greater than 12,000, AxisJava takes an average of 50% more time than .NET to respond.*

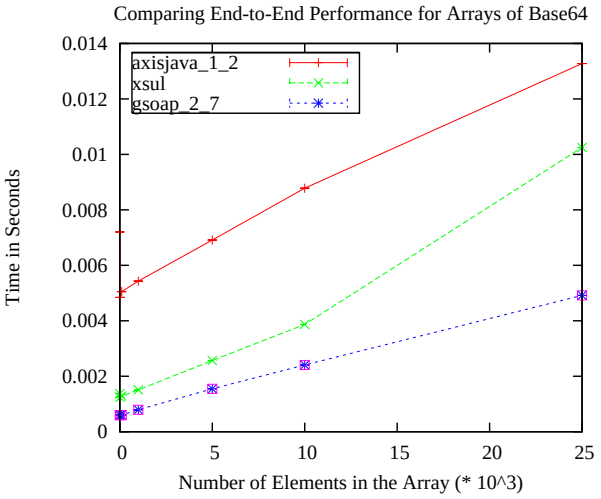

Figure 9. *This graph shows the performance results for end-to-end performance of binary data that has been base64 encoded (a feature provided by each of the toolkits). XSUL outperforms AxisJava, while gSOAP is consistently better than XSUL. For an array of size 10,000, XSUL is slower than gSOAP by a factor of 1.6, while AxisJava is slower than XSUL by a factor of 2.3.*

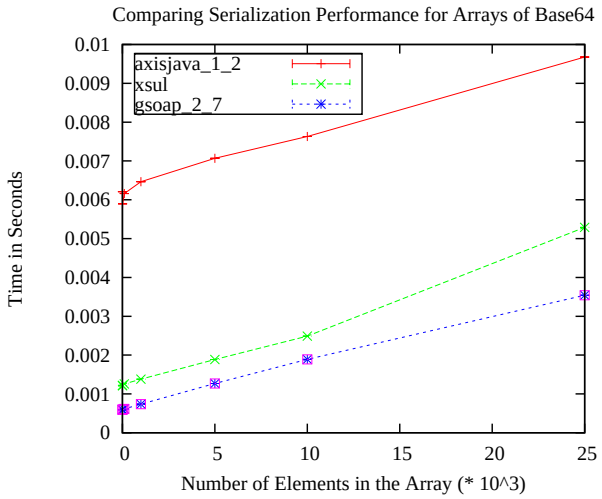

Figure 10. *This graphs shows the performance of AxisJava, gSOAP and XSUL for serializing data in Base64 format. AxisJava performs poorly compared to XSUL and gSOAP. XSUL takes 49% more time to complete than gSOAP for arrays of 25,000 elements.*

4 Observations on Current Toolkits based on Benchmark Results

We briefly describe the conclusions that can be drawn from our performance study of the current versions of five toolkits. The benchmarks and the associated drivers facilitate in repeating these tests for newer toolkits and to study the effect of improvements that are added to existing SOAP implementations.

- If low latency is critical, gSOAP is the ideal choice. On Windows, XSUL or .NET have comparable latency. AxisJava is not optimized for low latency requirements on either Windows or Linux.
- MCS [24] should use gSOAP or the .NET environment as these two toolkits scale well with increase in the size of complex data types. These toolkits can minimize the Web service overhead, which was identified as the primary bottleneck in the use of Web services with MCS. AxisJava, which is currently used, can severely hurt the scalability of the system.
- On Java-based Linux environments, XSUL should be used if arrays of primitives need to be sent or received. However, while XSUL can be used to send complex types, its performance does not scale well for receiving a sequence of complex data structures.
- For grid applications that repeatedly exchange data with similar structure, such as exchange of *ClassAds* be-

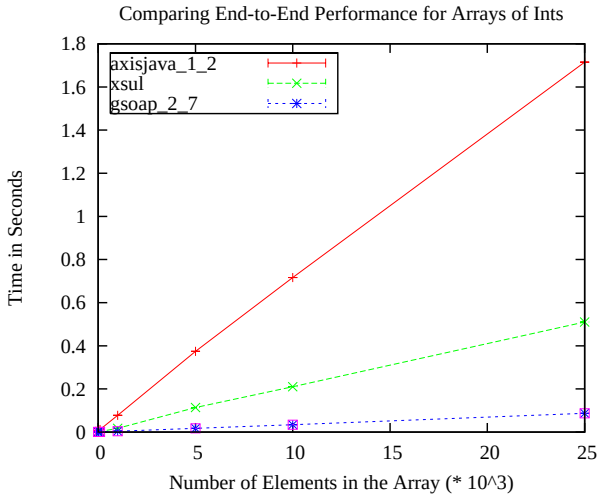

Figure 11. *In this graph we show the end-to-end performance for array of integers for AxisJava, XSUL and gSOAP toolkits. This requires each toolkit to deserialize, then re-serialize the input array. For an array of size 10,000 elements, AxisJava's time is a factor of 3.4 greater than XSUL's. For the same size, XSUL's time is a factor of 6.2 greater than that of gSOAP.*

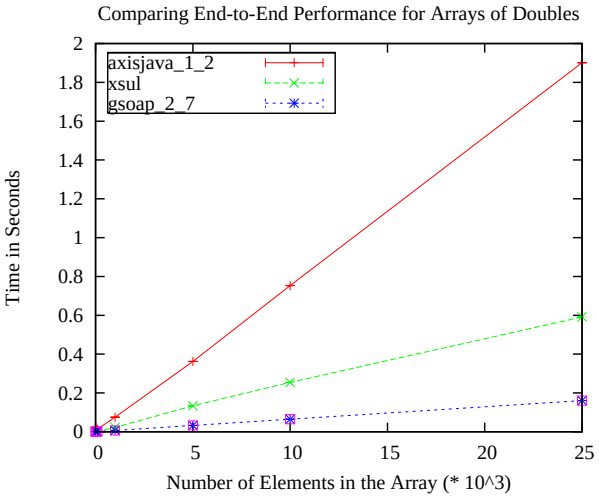

Figure 12. *Similar to Figure 11, but using doubles instead of integers, we compare end-to-end performance. For an array of size 10,000 elements, the time taken by AxisJava is a factor of 3.0 more than XSUL. For the same size, XSUL takes a factor of 3.9 times more than gSOAP.*

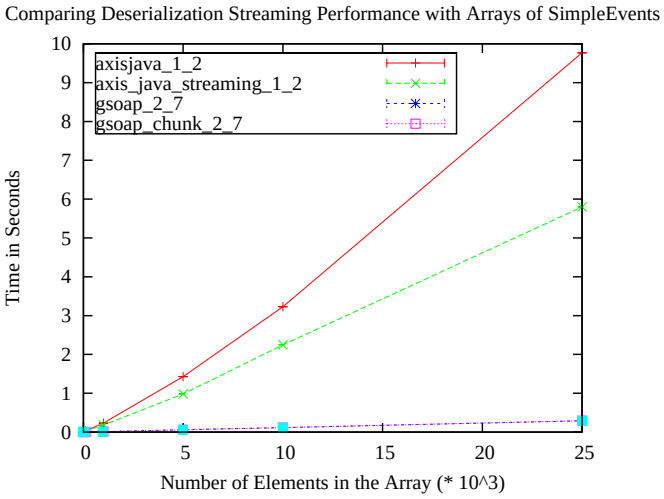

Figure 13. *We studied the effect of streaming by making AxisJava and gSOAP deserialize a large number of events with and without streaming enabled. AxisJava consistently performs better when streaming is used, taking 22% less time to complete for the 10,000 element array. gSOAP also trims its time with chunking on this test by 22%.*

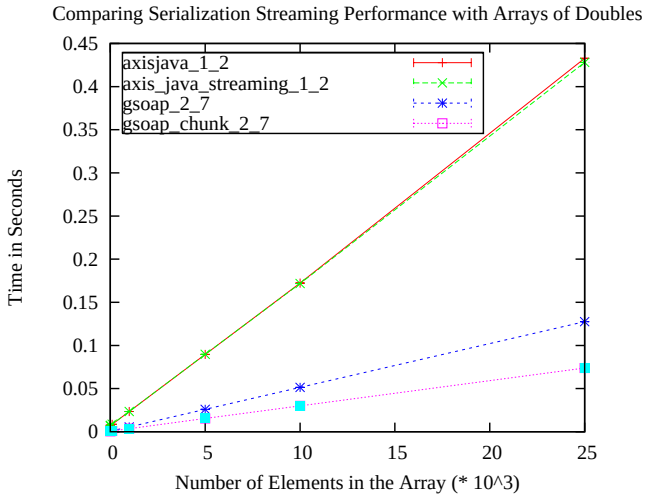

Figure 14. *We studied the effect of streaming by making AxisJava and gSOAP serialize arrays of doubles with and without streaming enabled. Unlike in the case of deserialization of events (see Figure 13), streaming does not improve AxisJava's serialization performance. Streaming helps improve gSOAP's performance for all sizes up to 25,000 elements, the maximum we tested for this benchmark. By enabling streaming, gSOAP's average response time for 10,000 elements was reduced by 42%.*

tween cluster managers in Flock of Condors [8], bSOAP performs extremely well. It also has impressive performance gains when only a small percentage of data changes during subsequent sends. bSOAP's caching mechanism is targeted towards such applications.

- bSOAP is comparable to gSOAP for sending arrays of doubles, integers and strings. However, for receiving data or end-to-end communication, gSOAP should be used.
- The use of streaming and chunking greatly improves the performance of gSOAP. With AxisJava, however, it makes minimal difference for serialization in AxisJava, though deserialization is helped. Whenever possible, persistent socket connections should be used for SOAP calls.
- On Windows, .NET is highly optimized for all the benchmarks. Its performance is comparable or better than XSUL for all data types. If available, .NET should be used instead of XSUL or AxisJava.
- The WSRF-Java implementation [13] uses the AxisJava toolkit. Our performance results show that XSUL, .NET or gSOAP based toolkits will be more efficient.

Axis Java is a widely-used toolkit under active development. While it performed poorly compared to the others in

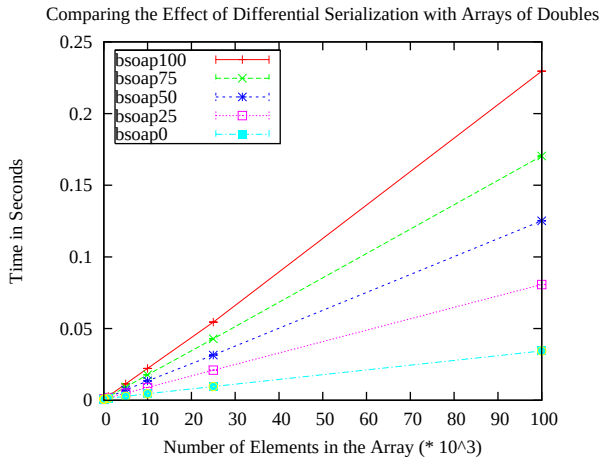

Figure 15. *Differential serialization is optimized for use-cases when subsequent sends are similar. In this benchmark. we measured the performance of bSOAP when 0% (bSOAP0), 25% (bSOAP25), 50% (bSOAP50), 75% (bSOAP75), and 100% (bSOAP100) of the values need to be re-serialized, rather than copied, for each message from the previous message. As expected, bSOAP's optimizations reduce the serialization time as the percentage of values that need to be re-serialized is reduced. For 100,000 doubles, the best case bSOAP0 is faster than the worst case bSOAP100 by a factor of 6.7.*

our benchmark tests, we expect to see improvements in the future.

5 Related Work

The SOAP community currently uses a set of well-known SOAP payloads and interfaces to test the *interoperability* of various toolkits [34]. Our work complements these efforts in that it aims to provide a standard set of workloads to test the various *features* and *performance characteristics* of SOAP implementations, rather than their interoperability.

The XMark project [3] has designed an XML benchmark suite to examine the performance of XML repositories, such as relational databases, for a wide range of queries that are typical of real-world application scenarios. This benchmark effectively compares different implementations of XML databases with queries that test specific primitives of the query processor and storage attributes.

In [22], the performance of SOAP is compared with CORBA and the Financial eXchange Protocol (FIX) [29], which is an established domain-specific protocol for capital markets. The motivation of this project is to study the applicability of SOAP for realistic business computing scenarios and the authors used real data from the Australian Stock Exchange for this purpose.

The performance of three different SOAP implementations (not named) are compared in [7]. Tests are conducted for business-oriented data such as account records and invoices. The authors conclude that the main bottleneck is the serialization and deserialization of messages, and that deserialization is generally more expensive than serialization for SOAP.

Karre and Elbaum [21] compare five Java-based XML parsers for well-formedness, validity and speed of parsing XML documents. They have developed a metric to determine the acceptance-accuracy and rejection-accuracy of each toolkit. For the performance study, they test the performance of parsing XML documents with varying number of tags. Our performance benchmark, in addition to including this test, also includes workloads that vary the nesting level for each document and the number of namespace qualified attributes in each element.

Our work complements the related work in performance evaluation and comparison of SOAP toolkits for business data, relational databases and interoperability tests.

6 Conclusions and Future Work

We motivated a strong need for SOAP benchmarks and described our benchmark suite, which effectively serves as a testbed for application programmers and Web services library developers to experiment, evaluate, and compare the performance of their toolkits. We demonstrated the benchmark suite's efficacy by providing a current performance snapshot for widely used toolkits. The description and use of the benchmark suite provide insights into relative strengths and weaknesses of various SOAP implementations.

Planned additions to the benchmarks include studying the performance for emerging security standards, once they stabilize. We will evaluate the performance results of AxisC++ and SOAP implementations in languages such as Perl and Python. We will also include benchmarks and automated tests to measure toolkit memory footprints, which will be important for embedded and hand-held devices.

References

[1] Amazon.com Web Service . `http://www.amazon.com/gp/aws/landing.html`.

[2] Google Web APIs. `http://www.google.com/apis/`.

[3] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, R. Busse . The XML Benchmark Project. Technical report, Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001.

[4] N. Abu-Ghazaleh, M. Govindaraju, and M. J. Lewis. Optimizing Performance of Web Services with Chunk-Overlaying and Pipelined-Send. *Proceedings of the International Conference on Internet Computing (ICIC)*, pages 482–485, June 2004.

[5] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju. Performance of Dynamic Resizing of Message Fields for Differential Serialization of SOAP Messages. *Proceedings of the International Symposium on Web Services and Applications*, pages 783–789, June 2004.

[6] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju. Differential Serialization for Optimized SOAP Performance. *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC-13)*, pages 55–64, June 2004, Honolulu, Hawaii.

[7] Alex Ng and Shiping Chen and Paul Greenfield. An Evaluation of Contemporary Commercial SOAP Implementations. In *Fifth Australasian Workshop on Software and System Architectures*.

[8] A. R. Butt, R. Zhang, and Y. C. Hu. A Self-Organizing Flock of Condors. *In Proceedings of Supercomputing 2003*, November 15-21, 2003, Phoenix, Arizona, USA. `http://www.sc-conference.org/sc2003/paperpdfs/pap265.pdf`.

[9] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the Limits of SOAP Performance for Scientific Computing. In *Proceedings of 11th IEEE International Symposium on High Performance Distributed Computing*, pages 246–254, Edinburgh, Scotland, July 23-26, 2002.

[10] Climate Research Committee (E. J. Barron, D. S. Battisti, B. A. Bov ille, K. Bryan, G. F. Carrier, R. D. Cess, R. E. Davis, M. Ghil, M. M. Hall, T. R. Karl, J. T. Kiehl, D. G. Martinson, C. L. Parkinson, B. Saltzman, R. P. Turco). Global ocean-atmosphere- land system (goals) for predicting seasonal-to-interannual climate. National Academy Press, Washington, D.C., 1994.

[11] E. Christensen et. al. Web Services Description Language (WSDL) 1.1, March 2001. `http://www.w3.org/TR/wsdl`.

[12] Global Grid Forum Working Draft. The Open Grid Services Infrastructure, 2003. `http://www-unix.globus.org/toolkit/draft-ggf-ogsi-gridservice-33_2003-06-27%.pdf`.

[13] Globus Alliance. Globus Toolkit Development Downloads . `http://www-unix.globus.org/toolkit/downloads/development/`.

[14] Globus Alliance, IBM and HP . The WS-Resource Framework, 2004. `http://www.globus.org/wsrf/`.

[15] Globus Toolkit. 3.9.4 Installer, 2005. `http://www-unix.globus.org/toolkit/downloads/development/`.

[16] M. Govindaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon. Requirements for and Evaluation of RMI Protocols for Scientific Computing. In *Proceedings of SuperComputing 2000*, November 2000.

[17] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, Canon, and H. F. Nielsen. Simple Object Access Protocol 1.1, June 2003. `http://www.w3.org/TR/SOAP`.

[18] IBM, Microsoft, Verisign . Web Services Security (WS-Security), April 2002.

[19] F. Illinca, J.-F. Hetu, and R. Bramley. Simulation of 3-d mold-filling and solidification processes on distributed memory parallel architectures. Proceedings of International Mechanical Engineering Congress & Exposition.

[20] Indiana University, Extreme! Computing Lab. Grid Web Services. `http://www.extreme.indiana.edu/xgws/`.

[21] S. Karre and S. Elbaum. An Empirical Assessment of XML Parsers. In *Sixth Workshop on Web Engineering*, pages 39–46, May 2002.

[22] C. Kohlhoff and R. Steele. Evaluating SOAP for High Performance Applications in Capital Markets. *Journal of Computer Systems, Science, and Engineering*, 63(4):(241–251), July 2004.

[23] Satoshi Shirasuna, Hidemoto Nakada, Satoshi Matsuoka, Satoshi Sekiguchi. Satoshi Shirasuna, Hidemoto Nakada, Satoshi Matsuoka, Satoshi Sekiguchi. In *Proceedings of 11th IEEE International Symposium on High Performance Distributed Computing*, pages 237–245, Edinburgh, Scotland, July 23-26, 2002.

[24] G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Mahohar, S. Pail, and L. Pearlman. A Metadata Catalog Service for Data Intensive Applications. *Proceedings of Supercomputing*, November 2003.

[25] A. Slominski, M. Govindaraju, D. Gannon, and R. Bramley. Design of an XML based Interoperable RMI System : SoapRMI C++/Java 1.1. In *Proceedings of PDPTA*, pages 1661–1667, June 25-28, 2001.

[26] SoapWare.org. The Leading Directory for SOAP 1.1 Developers. `http://www.soapware.org/directory/4/implementations`.

[27] Sun Microsystems. Dynamic Proxy Classes. `http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html`.

[28] The Apache Project. Axis Java. `http://ws.apache.org/axis/`.

[29] The Financial Information Exchange Protocol (FIX), version 4.3. FIX Protocol Ltd, August 2001.

[30] 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.

[31] R. van Engelen. Code generation techniques for developing light-weight efficient XML Web services for embedded devices. In *proceedings of 9th ACM Symposium on Applied Computing SAC 2004*, Nicosia, Cyprus, 2004.

[32] R. A. van Engelen and K. Gallivan. The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks. In *The Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002)*, pages 128-135, May 21-24, 2002, Berlin, Germany.

[33] W3C Note. SOAP Messages with Attachments. `http://www.w3.org/TR/2000/NOTE-SOAP-attachments-20001211`.

[34] XMethods.com. SOAPBuilders Interoperability Lab. `http://www.xmethods.com/ilab/`.