Differential Serialization for Optimized SOAP Performance

Nayef Abu-Ghazaleh, Michael J. Lewis, Madhusudhan Govindaraju
Department of Computer Science, Binghamton University
State University of New York
Binghamton NY 13902
{nayef, mlewis, mgovinda}@cs.binghamton.edu

Abstract

The SOAP protocol has emerged as a Web Service communication standard, providing simplicity, robustness, and extensibility. SOAP's relatively poor performance threatens to limit its usefulness, especially for high-performance scientific applications. The serialization of outgoing messages, which includes conversion of in-memory data types to XML-based string format and the packing of this data into message buffers, is a primary SOAP performance bottleneck. We describe the design and implementation of differential serialization, *a SOAP optimization technique that can help bypass the serialization step for messages similar to those previously sent by a SOAP client or previously returned by a SOAP-based Web Service. The approach requires no changes to the SOAP protocol. Our implementation and performance study demonstrate the technique's potential, showing a substantial performance improvement over widely used SOAP toolkits that do not employ the optimization. We identify several factors that determine the usefulness and applicability of differential serialization, present a set of techniques for increasing the situations in which it can be used, and explore the design space of the approach.*[1]

Key Words*: SOAP, Web Services, Serialization Optimization, High Performance, Scientific Computing*

1   Introduction

The Web Services model has recently been adopted as the basic architecture for Grid Systems [10]. Web Services provide standards for representing, discovering, and invoking services in wide area environments. The XML-based specifications, including the Web Service Description Language (WSDL) [9] and SOAP [15], provide extensibility and transparency. WSDL provides a precise description of a Web Service interface and of the communication protocols it supports, and SOAP is the most widely used communication protocol, facilitating the exchange of XML-based structured information. SOAP supports one-way messages, request-response interactions, peer-to-peer conversation, and RPC.

The characteristics that make SOAP attractive for the Grid include extensibility, language and platform independence, simplicity, robustness, and interoperability. Given the diverse nature of application requirements running on the Grid's heterogeneous computational components, SOAP is ideally suited to serve as a common standard protocol. However, since XML primarily uses ASCII as the representation format for data, sending scientifc data via standard implementations of SOAP can result in a severe performance penalty. It is important to identify and remove the bottlenecks in SOAP performance for scientific data. In this paper, we present a SOAP optimization technique that can result in significant performance improvements over widely used SOAP toolkits that do not employ the optimization, including gSOAP [24] and XSOAP [18, 21].

Applications of interest to the HPDC community often require communication using large arrays of floating point numbers and complex data types. Earlier work on SOAP performance identified the most critical bottleneck to be the conversion between floating point numbers and their ASCII representations [6]. The conversion routines account for 90% of the end-to-end message time. This paper introduces bSOAP, which addresses this serialization bottleneck. Rather than discarding serialized SOAP messages after they are sent, clients save the messages so they can be used as templates for future outcalls. Messages are completely serialized and saved during the first invocation of the SOAP call. Subsequent calls that are identical, or that have the same SOAP message structure, can avoid a significant percentage of the serialization overhead by requiring that only the *changes* to the previously sent message be seri-

alized. We call this technique *differential serialization*, and describe several techniques that make it effective, including:

- tracking data changes and overwriting only those values that have changed since the last send,
- expanding the serialized message to accommodate larger serialized values,
- storing the message in chunks and padding them with whitespace to reduce the cost of expansion, and
- overlaying the same memory region with different portions of the same outgoing message to reduce memory consumption.

We quantify the effectiveness of these techniques with a performance study that demonstrates that best case performance is up to ten times faster, for the case when messages can be resent in their entirety. We also show that send times can be reduced by a factor of five when only parts of the message need to be re-serialized. Our research is useful in two ways. Applications that repeatedly send similar messages will achieve significant performance improvement, and SOAP library developers will gain insights into the cases that make differential serialization most effective.

The remainder of this paper is organized as follows. Section 2 describes the SOAP protocol, and identifies and quantifies the serialization bottleneck. Section 3 describes the design and implementation of our approach. Section 4 contains a detailed performance study. We conclude with related and future work in Sections 5 and 6, and summarize our findings in Section 7.

2 Background: The SOAP Protocol

SOAP is a light-weight and extensible message exchange format. It is not tied to any specific programming language, platform, or transport mechanism, enabling the exchange of information across disparate run-time environments. Although HTTP is the most widely used transport layer for SOAP payload, other protocols such as FTP or SMTP can also be used. The use of XML and HTTP with the SOAP protocol makes it well suited to serve as an interoperable communication protocol on the Grid. It can be supported by many programming languages [22], including C, C++, Java, Perl, JavaScript and SmallTalk. SOAP is currently used in numerous Web Services based Grid toolkits. For example, the Java-based implementation in the GT3 [1] toolkit uses the Apache Axis SOAP implementation [23], the OGSA-C [12] implementation uses the gSOAP [24] toolkit, and XCAT [13, 19] uses XSOAP [21].

The serialization of SOAP calls can be logically separated into the following phases: (1) traversing the data structures of the invocation parameters; (2) translating the stored values into ASCII representations as required by the XML

specification; (3) copying the XML representation (including tags) into a buffer and (4) sending the buffer over the network. SOAP toolkits use various design strategies to implement these phases. The routines that convert data types (especially floats and doubles) to their ASCII formats can be complex and expensive. The design of the buffering mechanism can affect the number of system calls and cache hits in each serialization cycle. The choice of HTTP 1.0 or HTTP 1.1 can determine how the buffer is sent over the network. HTTP 1.1 supports chunking and streaming of messages allowing data structures to be sent over the network as soon as they are serialized.

In earlier work, we studied the performance of different stages of the SOAP implementation stack to isolate bottlenecks when various scientific data types are sent [6, 14]. The techniques for performance enhancement included the use of schema-specific parsing and trie data structures so that XML tags are parsed only once. We also studied the gain in performance due to the use of chunking and streaming. The test results indicated that these techniques affect only a fraction of the overall cost of a SOAP call. The most critical factor is the cost of conversion between floating point numbers and their ASCII representations. These conversion routines account for 90% of end-to-end time for a SOAP RPC call. For high performance applications, this bottleneck must be eliminated.

3 Differential Serialization: Design and Implementation

Our approach to removing the serialization bottleneck is to avoid complete serialization of SOAP messages by storing and reusing message templates. The idea is to perform a complete serialization only when the first message of a certain structure is sent by a SOAP communication endpoint. This message is then saved in the stub. Subsequent messages with the same structure and some of the same content (for example, calls to the same remote Web Service) can then reuse parts or all of the saved template instead of regenerating it from scratch. Although we focus our discussion and performance study on the client side, differential serialization could be used equally well by a server sending identical (or similar) responses to multiple separate clients.

In comparing an outgoing message to a saved template, there are four different matching possibilities:

Message Content Match: The entire message could be exactly the same as one that was sent from the client earlier. In this case, the client can simply resend the message as is, and avoid serialization altogether.

Perfect Structural Match: The structure and size of the message could be the same as an earlier message, but the values of some of the fields of the message could have changed. In this case, there is an opportunity to replace

the expensive serialization step with a faster step that writes only the changed values into the serialized buffer. The serialization of values that have not changed, and of the SOAP message metadata (tags) can be avoided.

Partial Structural Match: The structure of the message could be the same—that is, it could have the same header and field types—but some of the values *and the size of the message* may not match those of the saved template. Size mismatch results from the fact that, unlike in-memory base types, the serialized form of data can require different numbers of characters to represent. For example, encoding the integer 1 requires only one character, whereas 13902 requires five. In this case, the template could be expanded (or contracted) to meet the requirements of the new message. Performance improvement depends on how much faster it is to resize the message instead of serializing it from scratch.

First-Time Send: Finally, the first time a message is sent, it needs to be created (serialized) from scratch. The performance is the same as without differential serialization, plus the negligible overhead of checking to see if a stored copy exists and saving a pointer to it after it has been created.

These four cases provide the basis for our discussion. Clearly, message content matches provide the most opportunity for performance improvement, but only clients that send the same exact message repeatedly (to one or more different SOAP servers) can take advantage of it. The next best case is perfect structural matches, which don't require resizing the message template in memory. A *Data Update Tracking (DUT)* table tracks whether programs have changed data items since they were last serialized into the SOAP message. This allows us to limit the writing to only those values that have changed. We implement a technique that makes perfect structural matches more likely to occur (as opposed to the more expensive partial structural matches). We do this by *stuffing* serialized values with whitespace to accommodate potential future updates that would otherwise require expansion. To reduce the cost of partial structural matches, we store messages in potentially noncontiguous memory *chunks* to limit the impact of expansion, which could result in a substantial amount of expensive shifting and even memory reallocation. With message chunking, these effects are limited by the size of a chunk rather than the size of the whole message. Finally, we further reduce the cost of increasing field size by *stealing* extra space from neighboring fields, instead of shifting entire portions of message chunks.

3.1 Data Update Tracking (DUT) Table

When called upon to make an outcall, the client stub determines whether parts or all of the last copy of the same message type can be reused. To do so, the stub contains code that checks for a Message Content Match by using a DUT table, which associates in-memory data with their location in the serialized message template. Each saved message has its own DUT table, each of whose entries corresponds to a data element in the message, and contains the following fields:

- a pointer to a data structure that contains information about the data item's *type*, including the maximum size of its serialized form
- a *dirty bit* to indicate whether it has been changed since the last time the data was written into the serialized message
- a pointer to its current *location in the serialized message*
- its *serialized length*—the number of characters in the message necessary for storing the serialized form of the most-recently-written value
- its *field width*—the number of characters in the message template currently allocated to this data item (note that the field width must always match or exceed the serialized length)

If none of the dirty bits are set, the message has not changed and can be resent as is. Structural matches are implemented by scanning the DUT table and reserializing only those values whose dirty bits are set. Since DUT table entries point directly into the serialized form of the message, finding the location of the data item has constant cost. Clearly, this approach requires programmers to go through the DUT table when writing their in-memory data structures, and to be cognizant that the data they are using in memory will need to be serialized into a SOAP message. We foresee our SOAP library requiring all "serializable" data to be located in objects that contain "get" and "set" methods, whose implementation will update the DUT table transparently.

3.2 Shifting, Chunking, Stuffing, and Stealing

If the new serialized form of some value does not fit in the currently allocated space, we perform on-the-fly message expansion, which we call *shifting*. Shifting is necessary when the serialized form of the new value exceeds the field width value in the DUT table entry. At this point, all the bytes of the message are shifted to the right to make room for the new value, and the pointers into the message from other DUT table entries are updated accordingly.

To reduce the cost of shifting, serialized messages are not stored in contiguous memory regions; instead, we store them in variable sized potentially noncontiguous chunks. If necessary, chunks can be reallocated into different, larger memory regions, or split to form two smaller chunks. Configurable parameters determine the default initial chunk size, the threshold at which chunks are split into two, and the space that is initially left empty at the end of a chunk

(to allow for shifting without reallocation). Selecting the appropriate chunk size to reduce the cost of shifting must be balanced against several other factors that chunk size influences, including CPU cache effectiveness, the number of system calls needed to send messages (and whether the OS supports scatter-gather sends), the size of the underlying protocol implementation's send buffers, and the overhead of maintaining the message in chunks.

If we write into the serialized message a value that requires *less* space than the old value occupied, we simply rewrite the tag immediately to the right of the new value, and pad the space between the end tag of this field and the start tag of the next with whitespace, which is explicitly legal in XML (and therefore SOAP). This is one way that the field width can come to exceed serialized length for a data item. The other is by explicitly allocating more space than necessary when the first template message is generated. We call this *stuffing*. In particular, most types have associated with them a maximum number of characters that any of its serialized values can possibly occupy.[2] Setting field widths to maximum values can help avoid shifting altogether, at the expense of larger messages, both in memory and on the wire. Storing both the field width and the current serialized length, and allowing them to contain different values, also enables *stealing* space from neighboring data items instead of shifting entire portions of message chunks. This can further reduce the cost of expanding field sizes; we explore stealing in a separate paper [4].

3.3 Chunk Overlaying

Based on the description thus far, differential serialization has considerable memory requirements. In particular, it requires memory to store message data, the entire serialized form of the message, and the DUT table. Clearly this is not a desirable characteristic, especially as messages grow. *Chunk overlaying* helps limit memory requirements by allowing multiple portions of large arrays to be sent from the same message chunk. The approach takes advantage of the fact that large arrays contain multiple chunk-size portions that encode only the entries of the array. At any given time, the serialized data and the DUT table entries for only *one* portion of the array (a portion that will fit into a single chunk) is present in memory. That portion of the array is sent, and then the values of the next portion are serialized into the same chunk. This step requires that all the *values* (after the first chunk) be reserialized into the array. In addition to the known benefits of chunking and streaming (as used by HTTP 1.1 implementations), our approach has added potential performance gains because the tags that describe the data need not be rewritten. We explore chunk

---

[2]Note that strings cannot take advantage of stuffing because there is no maximum size string.

overlaying in a separate paper [3].

3.4 Applications that can Benefit

bSOAP is optimized for applications that resend similar messages repeatedly. The communication patterns of these applications determine the extent to which they can benefit from using bSOAP. A brief description of Grid applications that we think will be able to benefit from bSOAP, follows.

The Linear System Analyzer [11] is a high performance problem solving environment for large linear systems $Ax = b$. Its approach allows scientists to develop solution strategies by dynamically swapping out components that encapsulate linear algebra libraries. Scientists can connect various components in a cycle to repeatedly refine and recalculate the solution vector until the required convergence condition is met. Since the size and form of the array does not change over different iterations, consecutive messages exhibit perfect structural matches, so bSOAP could be used to achieve performance improvements.

The Metadata Catalog Service (MCS) [20] efficiently manages metadata associated with files generated by data-intensive applications. A general metadata schema is used to specify all the attributes associated with each file. MCS provides an API to perform various operations, including adding, deleting and querying metadata. Clients use SOAP to connect to the MCS Web service, which is connected to a backend MySQL database. Since each request sent by a user conforms to the metadata schema, the format of the SOAP payload is the same for each request. bSOAP perfect structural match can therefore be used to improve the performance of MCS.

Flocks of Condor systems [5] exchange ClassAd information to describe the resources in various Condor clusters that combine to define a large Grid-scale system. It stands to reason that information will be similar in structure and even content (if resource characteristics do not change) across multiple consecutive exchanges. Therefore, bSOAP would be able to automatically reserialize only the differences from previous exchanges, without requiring any alteration to Condor resource managers themselves.

Google and Amazon.com provide a Web services interface. The XML Schema used for the responses to user requests is always the same (for a particular operation in the Web service); only the values stored in the XML Schema instance change, because they depend on the queries sent by users. The optimizations in bSOAP for perfect structural match could significantly reduce the time spent serializing response messages from the heavily-used servers.

4 Performance Study

In this section, we describe the performance of our bSOAP implementation that uses differential serialization. The tests were run on a dual processor 2.0 GHz Pentium 4 Xeon with 1GB DDR Ram and a 15K RPM 18GB Ultra-160 SCSI drive running Debian Linux version 2.4.24. bSOAP and gSOAP code is compiled with gcc version 2.95.4 with optimization flag "-O2." We isolate and measure the *Send Time* in the client by starting a timer before preparing the message for sending, and stopping the timer right after the final *send()* system call on the socket. Relevant socket options, for both gSOAP and bSOAP, include SO_KEEPALIVE, TCP_NODELAY, SO_SNDBUF = 32768, and SO_RCVBUF = 32768. Because we're interested only in Send Time for this set of tests, each client connects to a dummy SOAP server on a different machine, over a Gigabit ethernet link; the server does not deserialize or parse the incoming SOAP packet. Our results reflect the average of 100 measurements for each reported data point.

4.1 Message Content Matches

This section studies the effect of the performance improvement, in the case where stored message templates can be reused without change. Thus, we characterize the performance improvement for message content matches. For these experiments, we vary the following factors:

- The *type of data* contained in the message. We have used integers, IEEE 754 standard doubles and *mesh interface objects (MIO's)*. An MIO is a structure of the form [int, int, double], where the first two fields represent mesh coordinates, and the third represents a field value. MIO's can be used, for example, for communication between two partial differential equation (PDE) solvers on different domains [17, 7].
- The *size of the message*: We vary message sizes by sending a single array containing 1, 100, 500, 1K, 10K, 50K, and 100K doubles.
- The *SOAP implementation*: We measure the performance of bSOAP with differential serialization turned on and turned off, and compare against unaltered implementations of gSOAP and XSOAP.

Figure 1 plots the average Send Time for SOAP messages of various sizes, containing a single array of MIO's. Figures 2 and 3 repeat the same tests for arrays of doubles and arrays of integers, respectively.

Figures 1, 2, and 3 show that bSOAP performance is slightly better than gSOAP, when both implementations serialize entire messages.[3] We compare our performance
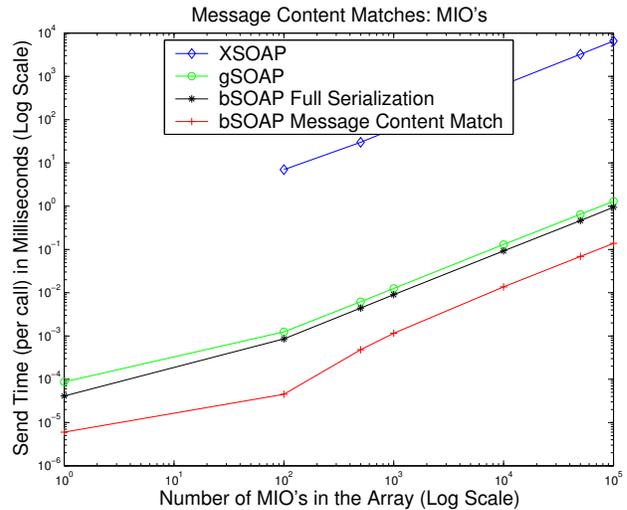

Figure 1. Comparing gSOAP and XSOAP to the Full Serialization of a bSOAP message, and to subsequent sends where the entire message is stored and can be resent without being changed ("bSOAP Message Content Match"). Send Time in milliseconds for various size arrays of MIO's. We have used a log scale on both the x-axis and y-axis.
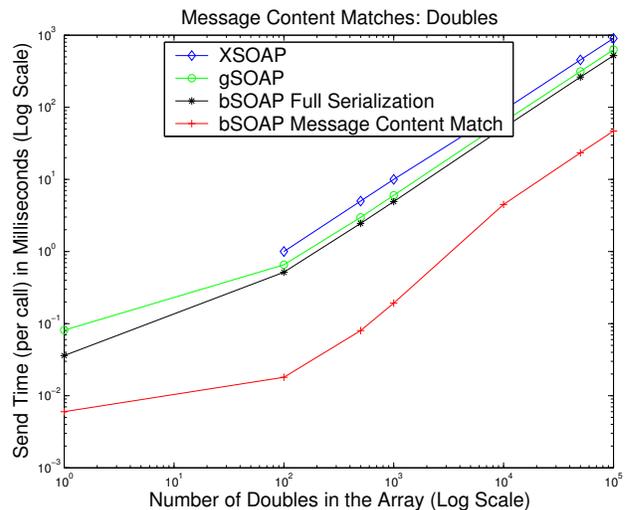

Figure 2. This figure corresponds exactly to Figure 1, for arrays of doubles instead of MIO's.

---

[3]gSOAP has full support for "multi-ref", bSOAP does not. We expect the performance of bSOAP with full serialization to be equivalent to that
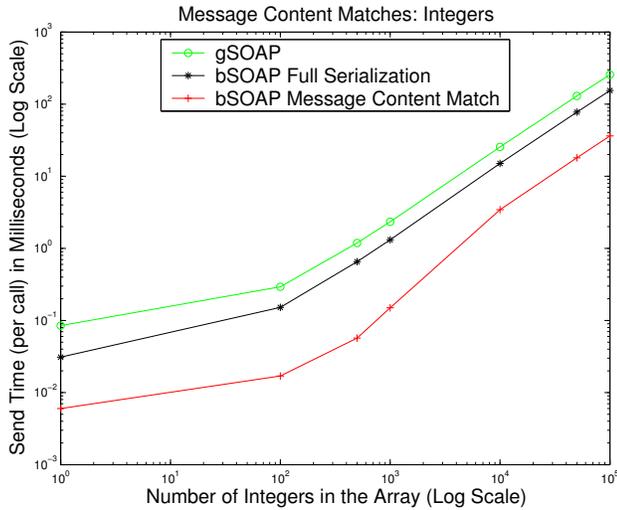
Figure 3. This figure corresponds exactly to Figures 1 and 2, for arrays of integers instead of MIO's or doubles.

against XSOAP, a fast Java SOAP implementation which, as expected, is still slower than C/C++-based gSOAP and bSOAP implementations. bSOAP message content matches are approximately seven times faster than full serialization for arrays of MIO's, approximately ten times faster for large arrays of doubles, and at least four times faster for large arrays of integers.

4.2   Structural Matches

This section explores the cost of writing data directly into a buffer rather than explicitly serializing messages on each send. That is, we characterize the potential performance benefit of perfect structural matches. Again, we vary the type of data sent, the size of the data, and the SOAP implementation. Our implementation with differential serialization varies the number of data items that need to be overwritten in the serialized version of the array. For this set of tests, we assume that the size of the array, and each of its elements, are the same in the template as they are in the new outgoing message, so shifting and stealing are unnecessary.

Figure 4 plots Send Time for various size arrays of MIO's. The graph re-plots bSOAP: Message Content Match and bSOAP: Full Serialization, from Figure 1. We also include bSOAP when 25%, 50%, 75%, and 100% of the MIO doubles must be re-serialized (the remaining portion stays the same as in the saved message, as do MIO integers). Figure 5 shows results of the same tests for doubles.

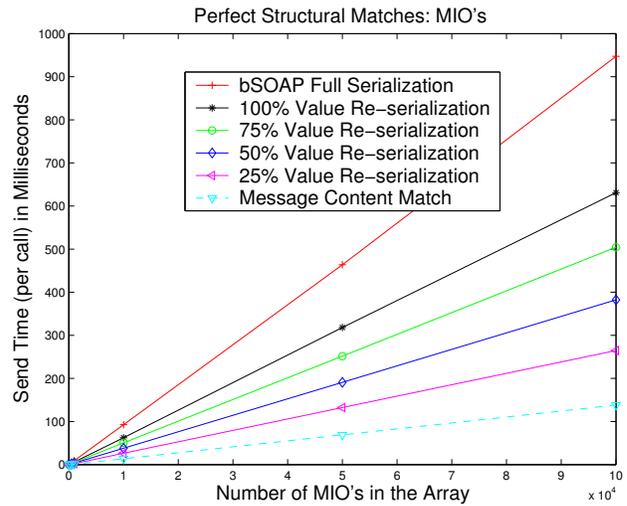of gSOAP when multi-ref support is added.

Figure 4. Send Time in milliseconds for various size arrays of MIO's, when various percentages of the stored values must be re-serialized.
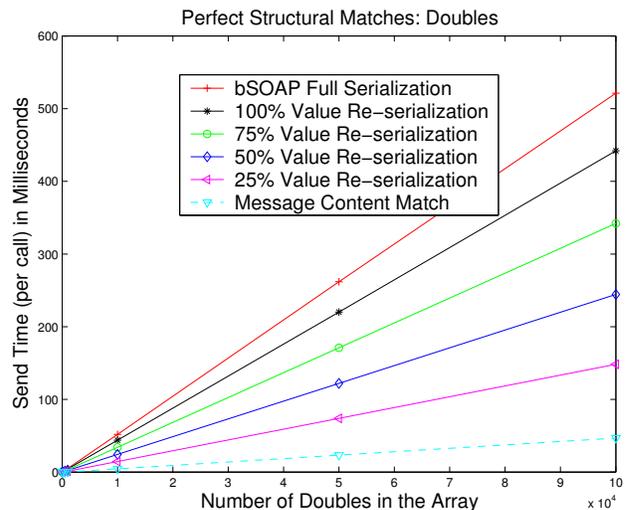

Figure 5. Send Time in milliseconds for various size arrays of doubles, when various percentages of the stored values must be re-serialized.
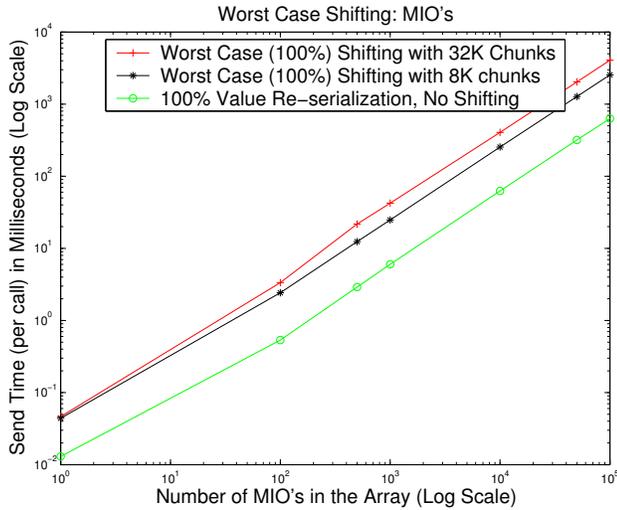
Figure 6. Send Time in milliseconds for various size arrays of MIO's. For worst case shifting, each value of the array must be expanded from the size of the smallest possible MIO (three characters) to the size of the largest possible MIO (46 characters).

Figure 7. Send Time in milliseconds for various size arrays of doubles. For worst case shifting, each value of the array must be expanded from the size of the smallest possible double (one character) to the size of the largest possible double (24 characters).

Figures 4 and 5 demonstrate that, as expected, Send Time depends directly on array size and on the percentage of values that must be re-serialized. The difference between 100% Value Re-serialization and Full Serialization shows the cost of generating and writing SOAP tags, compared to serializing only the data itself.

4.3 Shifting

This section quantifies the worst-case cost of shifting. Figure 6 shows the amount of time needed to insert largest size (46 character) MIO's into an array of smallest size doubles, causing shifting for each re-serialized value. Since shifting performance can depend on message chunk size, we ran the tests with a chunk size of both 8K and 32K. Figure 7 shows the results of repeating the tests with arrays of doubles.

Figures 6 and 7 show that shifting in the worst case can incur a significant performance penalty. In particular, Send Time when shifting all MIO's and doubles by the maximum possible amount is approximately four to five times slower when compared to re-serialization when shifting is unnecessary.

Fortunately, we don't expect the worst case to occur very often. Figures 8 and 9 plot Send Times for intermediate size values to maximum size values, when not all of the array values need to be re-serialized. These figures show that as the number of values that need to be re-serialized and shifted is reduced, the performance approaches the case
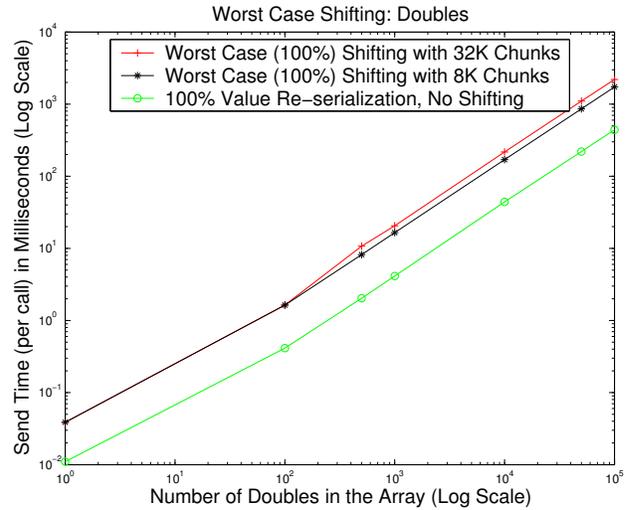
where shifting is unnecessary.

4.4 Stuffing

One way to avoid shifting altogether is to always allocate the maximum possible space for the value, and stuff to fill the unused portion with whitespace. For doubles the maximum encoded size is 24 characters plus the size of the tags, and for integers it is 11 characters, plus the size of the tags. There are two sources of overhead due to this approach. First, the client sends larger messages. To quantify the cost due to larger messages, we compared the cost of sending the smallest possible encoded values for doubles and MIO's (one and three characters respectively), with the cost of sending the same values within the maximum field size (24 and 46 characters). We also plot an intermediate field size for each (38 and 18 characters for MIO's and doubles). The results are shown in Figures 10 and 11, for arrays of MIO's and doubles, respectively.

The second source of overhead lies in shifting the closing tag when writing values that are smaller than those in the previous stored message. For example, when a large double is encoded, it consumes the full extent of the field size. When a smaller value is written on top of it for the next send, the closing tag must be written further left within the field, and whitespace must be written in the remainder of the field. To quantify this effect, we wrote smallest possible values for doubles and MIO's on top of largest possible
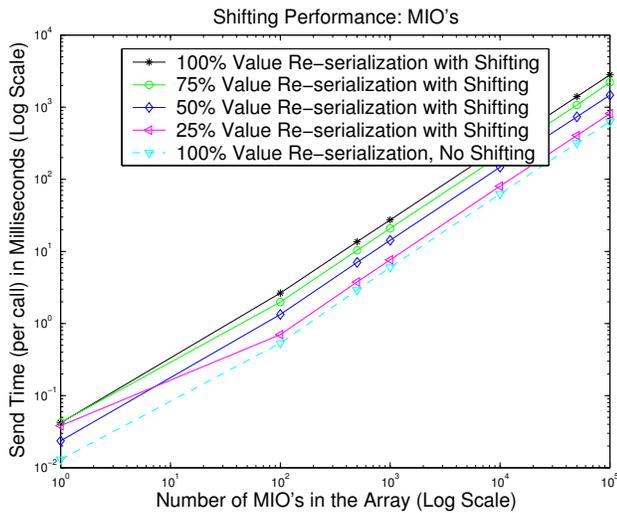
Figure 8. Send Time in milliseconds for various size arrays of MIO's, where different percentages of the array must be expanded from a 36-character MIO to the size of the largest possible MIO (46 characters).
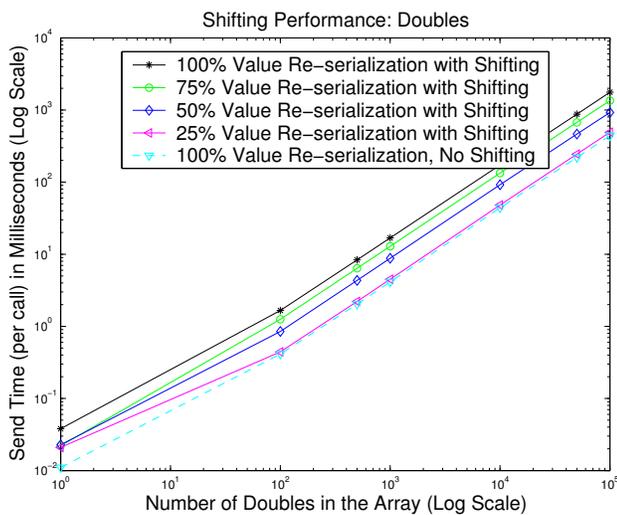

Figure 9. Send Time in milliseconds for various size arrays of doubles, where different percentages of the array must to be expanded from an 18 character double to the largest possible double (24 characters).
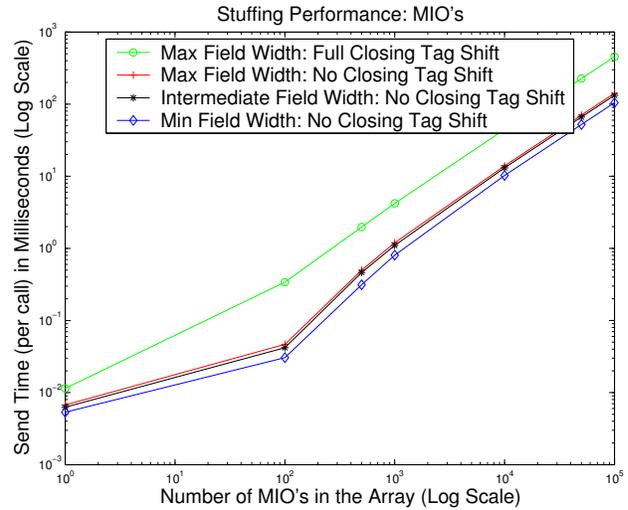

Figure 10. Send Time in milliseconds for various size arrays of MIO's, where values are stuffed to 46 characters (maximum width), 36 characters (intermediate width) and three characters (minimum width). Also plotted is the cost of writing three-character MIO's into fields containing 46 character MIO's, requiring a tag shift.

values; this results in the closing tag being shifted as much as possible. These plots are labelled as "Max Field Width: Full Closing Tag Shift" on Figures 10 and 11.

Figures 10, and 11 demonstrate that the most significant performance penalty of stuffing lies in shifting the closing tag rather than sending larger messages, for our worst-case tests. We expect this case to occur much less frequently than smaller tag shifts. This test was designed to reveal an upper bound on the performance penalty incurred by stuffing. However, writing single character doubles is less costly than writing larger doubles. Therefore, it is possible that the worst case lies somewhere between (a) writing the smallest double and the most whitespace, and (b) writing the largest double and no whitespace. Our current tests do not reveal where this worst case may actually lie.

4.5 Chunk Overlaying

To characterize the performance of chunk overlaying, we sent an array of doubles from a single 32K chunk of memory, and from separate 32K chunks of memory, all of which were in memory. With chunk overlaying, serialization of all values (except potentially some in the first chunk) is necessary, so we expect performance to be comparable to the 100% Value Re-serialization plot from Figure 5. Figure 12 confirms this hypothesis.
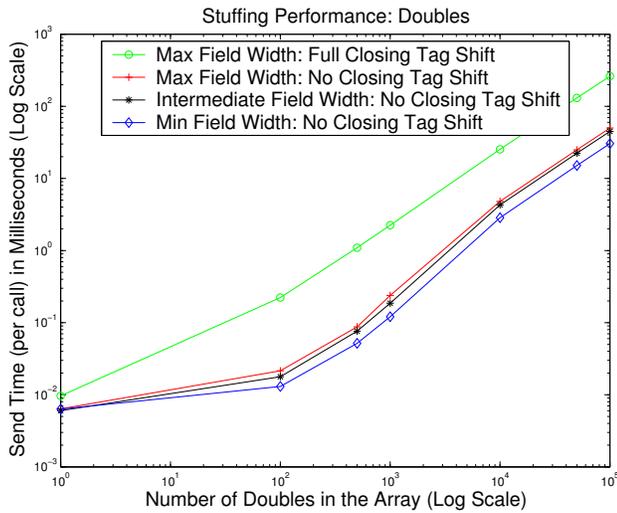
Figure 11. Send Time in milliseconds for various size arrays of one character doubles, where values are stuffed to 24 characters (maximum width), 18 characters (intermediate width), and one character (minimum width). Also plotted is the cost of writing single-character doubles into fields containing 24 character doubles, requiring a tag shift.
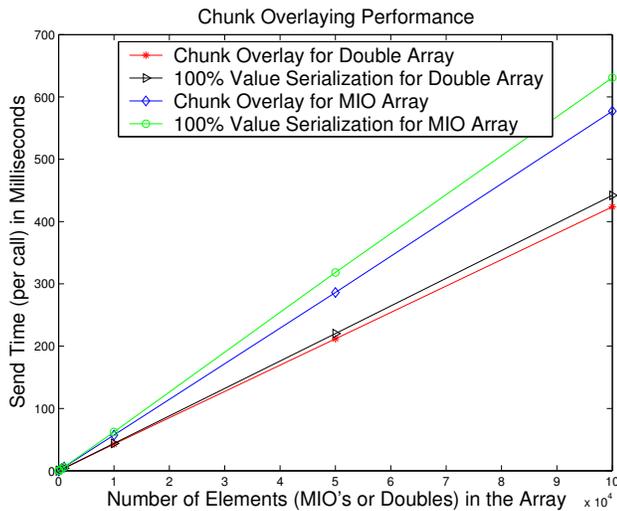

Figure 12. Send Time in milliseconds for various size arrays of MIO's and doubles, when sending from a single overlayed chunk vs. sending from multiple separate chunks.

5 Related Work

Chiuk et. al. [6] address SOAP performance bottlenecks by using trie data to reduce the number of comparisons for XML tags. This optimization is useful in SOAP deserialization, and is orthogonal to the issue of saving message templates. The other optimization they use is chunking and streaming of messages. gSOAP also provides this feature, in addition to compression, routing, and the use of optimized XML data representations using XML schema extensibility. These techniques are complementary to the ones we have proposed. They can be used when an RPC call must be serialized the first time; differential serialization can then be used for subsequent calls.

The SOAP specification allows the use of "multi-ref accessors"—identifiers that refer to previously serialized instances of specific elements of the SOAP call. Multi-ref accessors can be included within our serialized messages to further improve serialization performance.

Devaram et. al. [8] describe "parameterized client-side caching" of messages in files. Entire messages can be sent as is, and partial caching allows the client to reuse cached messages and change a few of the parameters for subsequent sends. The authors report a best case speedup of 800% over their own original code; this result is consistent with our speedup of approximately 1000%. However, the authors state that their approach is most appropriate for requests involving few parameters. The authors do not address how to apply their optimization to large arrays of scientific data (which we feel is the case where the technique is most useful), how to track which changes need to be made to the cached message, nor how to handle mismatched data sizes (requiring on-the-fly message expansion or stuffing).

The SOAP community has suggested several different specifications that would standardize SOAP binary formats, including base64 encoding, DIME [16] and BEEP [2]. While these techniques do achieve performance gains, they reduce the simplicity and universality of SOAP, the characteristic that makes it interoperable and attractive.

6 Future Work

Currently, each remote Web Service has its own saved template. For applications that send the same (or similar) data to different remote services, we plan to investigate the extent to which is would be beneficial for them to share message chunks across templates. This would allow serialization cost to be amortized across multiple sends to different Web Services. It also may be useful to store multiple different messages templates for the same remote service, rather than one per call type. We plan to quantify the effect that stuffing has on server-side decoding of incoming messages. Finally, storing messages at a SOAP server could help in

a completely different way, by suggesting the structure of future message arrivals. This could help avoid complete server-side parsing and improve performance, through differential *de*serialization.

7 Summary

We describe a new technique, called differential serialization, that helps alleviate the SOAP serialization bottleneck. Rather than reserializing each message from scratch, our approach saves a copy in the sender stub, tracks the changes that need to be made for the next message of the same type, and reuses this saved copy as a template for the next send. We describe techniques to increase the effectiveness and applicability of differential serialization, including on-the-fly message expansion, stuffing, message chunking, and chunk overlaying. For applications that resend the same messages repeatedly, our performance study demonstrates an improvement in Send Time by a factor of four to ten for arrays of different types of data. We also show that resending messages with similar structure but containing some different values can also achieve significant speedup. We characterize the performance penalty for on-the-fly message expansion, and describe several techniques for counteracting its adverse effect, including stuffing and chunk overlaying.

References

[1] Globus Toolkit 3.0.2. `http://www-unix.globus.org/toolkit/download.html`.

[2] The Blocks Extensible Exchange Protocol Core (BEEP), March 2001. `http://www.ietf.org/rfc/rfc3080.txt`.

[3] N. Abu-Ghazaleh, M. Govindaraju, and M. J. Lewis. Optimizing Performance of Web Services with Chunk-Overlaying and Pipelined-Send. *To appear in the International Conference on Internet Computing (ICIC)*, June 2004.

[4] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju. Performance of Dynamic Resizing of Message Fields for Differential Serialization of SOAP Messages. *To appear in the International Symposium on Web Services and Applications*, June 2004.

[5] A. R. Butt, R. Zhang, and Y. C. Hu. A Self-Organizing Flock of Condors. *SC '03*, November 15-21, 2003, Phoenix, Arizona, USA. `http://www.sc-conference.org/sc2003/paperpdfs/pap265.pdf`.

[6] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the Limits of SOAP Performance for Scientific Computing. In *Proceedings of HPDC-11*, pages 246–254, Edinburgh, Scotland, July 23-26, 2002.

[7] Climate Research Committee. Global Ocean-Atmosphere Land System (GOALS) for Predicting Seasonal-to-Interannual Climate. National Academy Press, Washington, D.C., 1994.

[8] K. Devaram and D. Andresen. SOAP Optimization via Parameterized Client-Side Caching. In *Proceedings of PDCS 2003*, pages 785–790, November 3-5, 2003.

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

[10] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *Computer 35(6)*, 2002.

[11] D. Gannon, R. Bramley, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Breg, S. Diwan, and M. Govindaraju. *Enabling Technologies for Computational Science*, chapter 10 The Linear System Analyzer, pages 123–134. Kluwer, Boston, 2000.

[12] Globus Alliance. OGSA-C. `http://www-unix.globus.org/ftppub/ogsa-c/packages/`.

[13] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley. Merging the CCA Component Model with the OGSI Framework. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 12-15, 2003, Tokyo, Japan.

[14] 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.

[15] 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`.

[16] IBM and Microsoft Corporation. Direct Internet Message Encapsulation (DIME). `http://www-106.ibm.com/developerworks/library/ws-dime/`.

[17] F. Illinica, J. F. Hetu, and R. Bramley. Simulation of 3D Mold-Filling and Solidification Processes on Distributed Memory Parallel Architectures. In *Proceedings of International Mechanical Engineering Congress and Exposition*.

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

[19] S. Krishnan and D. Gannon. XCAT3: A Framework for CCA Components as OGSA Services. In *Proceedings of HIPS 2004: 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, April 2004.

[20] 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.

[21] 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.

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

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

[24] 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.