

SOAP-binQ: High-Performance SOAP with Continuous Quality Management

Balasubramanian Seshasayee, Karsten Schwan, and Patrick Widener
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

{bala, schwan, pmw}@cc.gatech.edu *

Abstract

There is substantial interest in using SOAP (Simple Object Access Protocol) in distributed applications' inter-process communications due to its promise of universal interoperability. The utility of SOAP is limited, however, by its inefficient implementation. Our research aims to make SOAP useful for high end or resource-constrained applications. The resulting SOAP-bin communication protocol exhibits substantially improved performance compared to regular SOAP communications. Gains are particularly evident when the same types of parameters are exchanged repeatedly, examples including transactional applications, remote graphics or visualization, and distributed scientific codes. A further improvement to SOAP-bin, termed SOAP-binQ, addresses resource-constrained applications like distributed media codes, where scarce communication bandwidth, for example, may prevent end users from interacting in real-time. SOAP-binQ offers quality management functions that permit SOAP to reduce parameter sizes dynamically, as and when needed. The methods used in size reduction are provided by end users and/or by applications, thereby enabling domain-specific tradeoffs in quality vs. performance. An adaptive use of SOAP-binQ's quality management techniques presented in this paper significantly reduces the jitter experienced in two sample applications like remote sensing and remote visualization.

1. Introduction

SOAP (Simple Object Access Protocol) is an XML-based remote invocation protocol designed for flexibly composing Internet applications [18]. Its use of text-based messaging (XML) and of HTTP or SMTP for communication

provides universality and interoperability, but also creates substantial performance limitations. This reduces the utility of SOAP for large classes of applications. An example is the use of wide-area networks (including from remote sites) by amateur astronomers to share their captured images and track, in real-time, low-light emitting objects like asteroids[19]. Other examples include high end business applications like the operational information systems presented in [8] and real-time collaborations in which scientists jointly visualize simulation or sensor data to better understand or interpret certain phenomena[21, 7, 12]. These applications have in common the use of substantial computation in conjunction with large-data communications, in the presence of limited networking and/or CPU resources. Computations include data selection, filtering, and transformation, as exemplified by image processing pipelines in sensor systems and by the graph-structured data manipulations used in scientific collaboration (e.g., to transform scientific data to match a specific end user's needs[13]).

In all of the applications cited above, SOAP's use of XML with its ASCII-based parameter representations can be prohibitively expensive. Costs include server loads due to the need to manipulate and translate ASCII data and communication overheads due to the relatively 'bulky' ASCII representations of invocation parameters.

This paper describes how to attain high performance and low overhead for SOAP-based communications. The approach taken leverages work on adaptable object communications and recent results on efficient binary representations for XML-based data [20, 6]. Specifically, the idea is (1) to associate with SOAP communications application-specific handlers that can be used to manipulate parameters used for invocations and transform parameters into more suitable formats, and (2) to make such handlers efficiently executable, even for large data volumes, by using binary representations for both the handlers and the XML data they manipulate. The outcomes are the SOAP-bin and SOAP-binQ protocols described next.

The basic SOAP-bin protocol uses *conversion handlers*

* This work is funded in part by the National Science Foundation's ACIR program.

to implement XML-to-binary conversions, if needed. It can be used in multiple ways:

- *SOAP-bin - high performance mode*: SOAP-bin can be used to implement server-based communications, as with the internal communications used in web server front-to-backend communications. In such ‘internal’ communications, SOAP parameters never appear in XML forms. Instead, they have already been converted to corresponding binary forms, for request and for result parameters. Measurements presented in this paper demonstrate that the performance of SOAP-bin used in this mode is competitive with the standard invocation mechanisms used in today’s client-server interactions, like Sun RPC.
- *SOAP-bin - interoperability mode*: when servers receive requests from and return data to external clients, clients use standard XML data, but servers use binary data, in order to reduce server loads. Measurements presented in this paper demonstrate that the required ‘one-sided’, just-in-time, client-side data conversions permit SOAP-bin to outperform standard implementations of SOAP, with respect to the communication bandwidth consumed (in particular for wide area links).
- *SOAP-bin - compatibility mode*: the lowest performance case for SOAP-bin is where both end users, such as clients operating in peer-to-peer mode, must translate XML text to binary data, in order to be able to operate on such data with standard tools. Adobe’s PDF format and image formats used by graphics tools are typical examples. To evaluate such costs, we present measurements in which XML data is first converted to binary form, transferred, then converted back to XML, and we compare the performance of SOAP-bin in this mode with the performance of regular SOAP and of SOAP that uses online compression to reduce data size. Both communication overheads and client loads are reported.

An enhanced version of SOAP-bin, termed SOAP-binQ(uality), permits applications to enrich basic conversion handlers with configurable, application-specific functionality. The resulting *quality handlers* are code modules that take as inputs both the binary representations of SOAP parameters and *quality attributes* that determine handlers’ behaviors. Attributes may be changed on a per invocation basis, and they may be provided by applications and/or by the underlying network/system levels. An example from the scientific domain is an application-provided data filter that adjusts the amounts (and therefore, the quality – in terms of resolution) of the data sent to the current needs of clients and/or also to currently available network resources. Such client- and network-aware data

filtering has been shown useful for a variety of applications and platforms, including to control the data volumes required for remote 3D visualizations across the Internet [10].

In measurements presented in this paper, we apply configurable quality handlers to filter both sensor data and scientific data, and we demonstrate the performance improvements derived from the adaptive use of such filters for end user applications. Improvements are due to the improved ability of SOAP-binQ compared to SOAP-bin to control the data volumes exchanged as call or return parameters, thereby better dealing with runtime variations in communication and server resources.

Initial results are encouraging. With the SOAP-binQ infrastructure in place, message transmission times are improved by a factor of about 10 for 1MByte message sizes. Marshalling and unmarshalling times and therefore, the loads SOAP imposes on server systems are also reduced significantly. Both improvements are due to the use of binary formats for transporting SOAP parameters. Finally, distributed large-data applications experience more uniform response times in congested networks when using SOAP-binQ, due to its ability to dynamically adjust the data volumes sent to available network resources.

The remainder of this paper elaborates on the design, implementation and behavior of SOAP-binQ and compares it with XML-based SOAP implementations. Related work is described in Section 2. Section 3 deals with the overall design and implementation details. Experimental results accompanied by discussions are presented in Section 4. Conclusions and future work appear in Section 5.

2. Related Work

The growth of XML-based web services is increasing the interest of high performance end users in these technologies. Efficiency issues with XML and with the SOAP protocol that relies on it, however, have inhibited technology adoption. Inefficiencies with XML are described in detail in [2], which identifies ASCII conversions of digits as one of the most significant bottlenecks associated with XML. An approach first demonstrated in [20] is to use binary encodings of XML for large-data objects [2], with resulting performance improvements of up to 75% in terms of reductions in processing costs and message sizes achieved in [17]. We adopt this approach [20], by automatically converting XML schema-based data descriptions to binary forms (and vice versa) [6], ‘just in time’ (i.e., when needed by end user applications). Use of binary data in place of XML is also mentioned in [1]. [15] shows how binary data can be transported using XML.

Interactive scientific applications, remote instrument usage, remote sensing, and multi-media applications typically

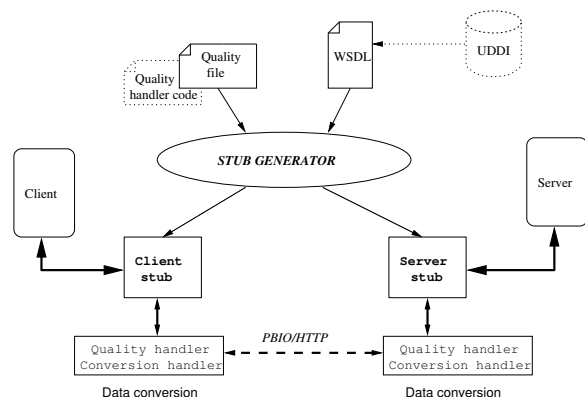


Figure 1. Software Architecture of SOAP-binQ.

require quality of service (QoS) support. The QoS mechanisms defined for SOAP (version 1.2) focus on QoS guarantees provided by the transport level and/or by intermediaries involved in SOAP processing. In accordance with ongoing research on QoS in middleware [10], our approach generalizes such transport-focused QoS to enable applications to directly specify and manipulate the data sent and received by SOAP participants. This permits application-specific tradeoffs in the amounts and therefore, quality of information sent and received vs. the available network and processing resources on participating machines. The contract-based quality support used in SOAP-binQ is conceptually similar to policies in [11].

3. Software Architecture of SOAP-binQ

3.1. Architecture Description

Scientific computing in distributed environments must efficiently deal with data heterogeneity, at the level of machine architectures and for applications. In scientific visualizations, for instance, the data produced by an application must typically undergo a sequence of transformations before it is displayed to a particular end user [13]. Soap-binQ deals with data heterogeneity by *describing* SOAP parameters with XML, and by *operating* only on the binary representations of such data. The intent is to eliminate or reduce serialization and deserialization costs, while also maintaining the universality implied by the use of XML.

Figure 1 shows the overall design of SOAP-binQ. It consists of a WSDL compiler that generates the client and server side stubs, with conversion handlers for XML/binary inter-conversion. Quality attributes are specified in a *quality file*, which is compiled jointly with the WSDL file to generate stub files. The information contained in this file are the data types of the parameters sent in SOAP messages in

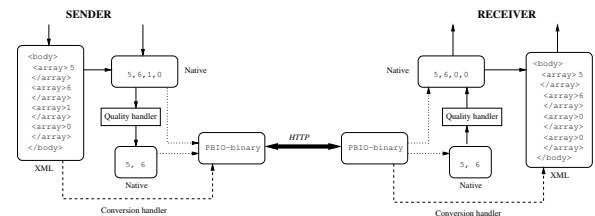


Figure 2. Data flow in Soap-binQ communications.

conjunction with various quality attribute values. It also references the quality handlers specified by end users (when present) or generates trivial quality handlers otherwise.

Figure 2 illustrates the transformations a message can undergo during a simple SOAP request/response interaction. The figure also depicts the different options available for such interactions. These options exist because the application layer can communicate with the transport layer either via XML data or via native data. In the first case, data is converted to native form. Once this is done, handlers can perform data transformations or filtering. This is discussed in more detail in the following paragraphs. The data format used for data transmission via an HTTP connection is PBIO[6](described in Section 3.2). It deals with heterogeneous machine architectures via a ‘receiver makes right’ paradigm, thereby avoiding the symmetric up and down translation used in most grid middleware. At the receiving end, the data is provided to the application layer in the appropriate format. In previous work, PBIO has been shown to efficiently represent the structured XML that constitute SOAP parameters[20]. We have developed and characterized SOAP-bin on HTTP transport. SMTP is not considered, since it requires binary messages to be base64 encoded, thus making it unsuitable for high performance applications.

3.2. SOAP-binQ Implementation

SOAP-bin. Our prototype implementation of SOAP-binQ is based on *Soup* [9], a C version of SOAP developed by Ximian. Soup is modified to read in both a handler file and a WSDL file, and from these files, produce the modified stubs and support files that implement the desired quality management functionality. Specifically, the WSDL compiler reads XML typecodes from the WSDL file, generates the client-side and server-side stubs as well as a file with support functions and a header. The server and client side applications are compiled along with the stubs. The schema used in Soup identifies the basic types as integer, char, string and float, and it allows the user to build more complex types through the use of lists and structs. Soup uses *libxml2* for conversion from and to XML. Our implementation alters

this conversion to use the P BIO binary data format. Expat toolkit [3] is used for parsing XML.

P BIO[6] is a binary representation that allows the sender to send data in its native format. It uses dynamic code generation to automatically generate code that performs the format conversion at the receiver side. P BIO data is defined through what is known as formats. Formats are similar to XML schemas, in that they define how data is structured. Every P BIO transaction begins with a registration of the format with a “format server”, which collects and caches P BIO formats. Whenever a new type is encountered, the application consults the format server to interpret the message. Interpretation of the incoming format is carried out through dynamically generated code. If the sender and the receiver have the same architecture, no conversion is necessary (this optimization is also used in [4]). Any format thus obtained is cached locally, hence the transaction with the format server occurs only once per format.

SOAP-binQ: Quality Files. The purpose of SOAP-binQ’s quality support is to allow transformations to be applied to messages, when matching message sizes to available resources. Since many applications face tradeoffs between data accuracy and speed, it is useful to provide transport-level support for varying the volume of data exchanged between participants. SOAP-binQ implements this by permitting developers or end users to create a ‘quality’ file accompanying the WSDL information. This file dictates the data precision to be used under differing resource availabilities. For instance, data with a specified number of array values could be replaced by a smaller sized array, if the loss in precision is not as critical as the time that could be lost serializing, transmitting and deserializing a larger array. As an example, in this paper, we demonstrate the tradeoffs existing for an image server used in astronomy applications. Here, the resolution of the image is adjusted based on the quality of the network link. Network quality is represented by the cumulative RTT values for SOAP requests.

Quality files are created by end users or provided by domain-knowledgeable developers. Such files contain handlers and the QoS attributes in the form of intervals of quality attributes. The policy-level information encoded in quality files is formulated based on the requirements of the application. Since this information consists of different data formats used in messaging, the data types involved in messaging must be known when the quality file is created. In the future, we foresee the designer providing a quality file along with the WSDL file, through UDDI or a similar WSDL repository. This would let the user directly access the service, without knowledge of the actual message types used in data transmission. In designing the system, we decided to let the application to expose its data structures to the transportation layer via an extension to WSDL, rather than exposing the lower layer to the application through API calls

and allowing the application to manage the quality. This way, existing applications can easily be used with our protocol without any changes to the applications themselves.

Quality Attributes. Quality files relate quality attributes to message types, where RTT is used as the monitored value in our current examples. However, a monitored attribute can use any value that is suitable for triggering changes in data quality, such as desired image resolution. Other attributes suitable for the sample applications and execution environments used in this paper may capture CPU load, memory consumption, or similar factors. The reader is referred to [14] for a more detailed discussion of how to link monitored attributes to changes in the quality of service (QoS) or quality of information (QoI) in quality-managed applications.

Dynamic Quality Changes. The quality file draws the correspondence between quality attributes and message types in the stubs, but it may be necessary for the application to change quality management at runtime. Our current implementation does not permit runtime changes in the handlers or policies used for quality management, but it does permit applications to dynamically update the values of quality attributes. Note that in all such cases, while the messages to be included in the quality policy are determined by studying the application’s needs, performance testing is required to determine suitable values of quality attributes.

4. Experimental Evaluation

We illustrate the performance of SOAP-binQ through measurements obtained in the following experiments. First, we demonstrate the performance of SOAP-bin by comparing it with Sun RPC (which uses the XDR data representation). The intent is to show that the basic performance of SOAP-bin is competitive with that of standard client-server communication mechanisms. Second, we systematically characterize the operational costs of binary SOAP and compare it with XML-based and with compressed XML implementations of SOAP. Next, SOAP-bin’s three performance modes are compared using the above results. Lastly, the performance of SOAP-binQ and the utility of runtime quality management are evaluated with representative applications.

For each experiment, we conduct a series of measurements to evaluate the performance of SOAP-bin with different data types and sizes. Two sets of data types are used as benchmarks, one representing scientific applications via integer arrays of different sizes, and a second representing business applications via a nested struct of varying depth. Arrays are at one end of the spectrum, where marshalling simply means enumerating the elements - enclosing them with tags in the case of XML and simple enumeration in the case of P BIO. At the other end of the spectrum lie nested

structs, since marshalling and unmarshalling require recursive function calls and the addition of tags (in the case of PBIO, this involves traversing the struct and copying the fields to a send buffer).

4.1. Sun RPC vs SOAP-bin

The overall performance of SOAP-bin is compared to TCP-based Sun RPC. The results of the experiments (with methodology, detailed in [16]), conducted over 6 different sizes of arrays and nested structs demonstrate that Sun RPC outperforms SOAP-bin by about a factor of 1.8 in the case of arrays and 3.9 in the case of nested structs, on an average. The difference is due to SOAP-bin's use of HTTP for its transactions, and the relatively high cost of PBIO encoding/decoding for highly nested structs. It can be concluded that, while SOAP-bin cannot be a replacement for Sun RPC, the performance attained is reasonable for Internet-based high performance applications. However, since PBIO has the added advantage that the sender can issue data in its native binary format, servers dealing with PBIO data can scale better, especially for large data sizes[6].

4.2. SOAP-bin: Microbenchmarks

SOAP-bin: Marshalling/Unmarshalling Costs. A set of experiments were conducted to isolate the costs of marshalling, unmarshalling and transmission. Experiments use different network links, one representing a high end link in a company's intranet and the other a low end, remote Internet link. The high end link is a 100Mbps LAN link in one of our laboratories; the low end link connects a laboratory machine to a home machine via ADSL. In both cases, the client machines are 2.2GHz Pentium IV with 512MB RAM running Linux kernel 2.4.18-27.7.x. The server has the same configuration as the client in the case of the 100Mbps link, but it is a 1.9GHz Pentium IV with 512MB RAM, running Linux kernel 2.4.18-3 for the ADSL link. Measurements are derived from sets of 10-1000 experiments, reporting the averages over all readings, after discarding the first sets (to eliminate cold start effects). Variances are less than 1% on the average and are therefore, not reported. Compression is achieved using Lempel-Ziv encoding. Compressed XML is mostly the same size as, and sometimes smaller than the equivalent PBIO data, whereas the XML data is about 4-5 times the size of the PBIO message for arrays and about 9 times for nested structs.

The time taken for PBIO encoding and decoding is relatively small when compared to data transmission costs, especially with larger data sizes. This is due to the fast nature of the participating machines, making communication latency the restricting factor. This effect is more pronounced in the case of a slower connection, the ADSL, where the gap

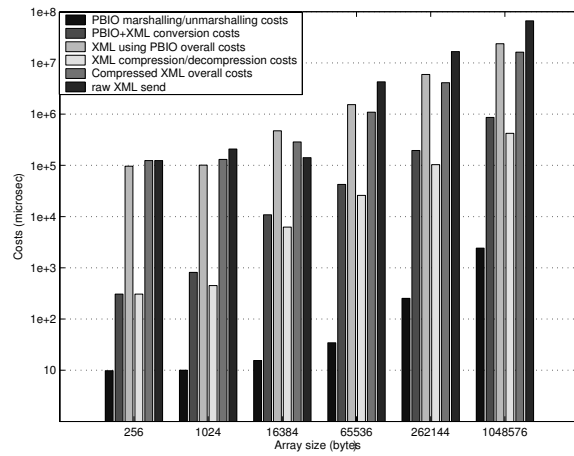


Figure 3. Comparison of SOAP-bin costs with XML compression and direct send for integer arrays over an ADSL link.

between PBIO encoding/decoding and the transport costs is quite high, even in a log scale graph.

SOAP-bin: Costs with XML data. A second set of experiments conducted with the same setup assumes that data is available to the transport in XML rather than in native format. A typical scenario is the use of SOAP by an end user application interacting with a database, for instance. In such cases, additional work is needed to perform data conversion from and to XML.

Experimental results in Figure 3 show the various costs involved in SOAP, SOAP-binQ and SOAP with compressed XML, for integer arrays of various sizes, over an ADSL link (results of the same experiments over 100 Mbps link and for nested structs have been reported in [16]). The advantages of using binary data encodings for SOAP parameter transmission are less evident in this case, principally due to the need to explicitly parse XML. In addition, we have not optimized the XML-PBIO conversion handlers. In the case of the 100Mbps link, for instance, data conversion takes more time than simply sending raw XML, for both the array data and the nested structs. In contrast, with the ADSL link (peak bandwidth of about 1Mbps), XML-PBIO conversion has clear advantages, especially for large sizes, since this conversion is equivalent to compressing XML to about 1/4 of its original size. Apparently, it is more advantageous to compress XML, as evident from the fact that this method is the fastest. However, compression is not a suitable choice if the application at either end produces or consumes data in binary form (as with typical large-scale web servers that use backend machines). Secondly, in a client-server architecture, the server typically sends large data as response, and it is important to keep computation at the server side at

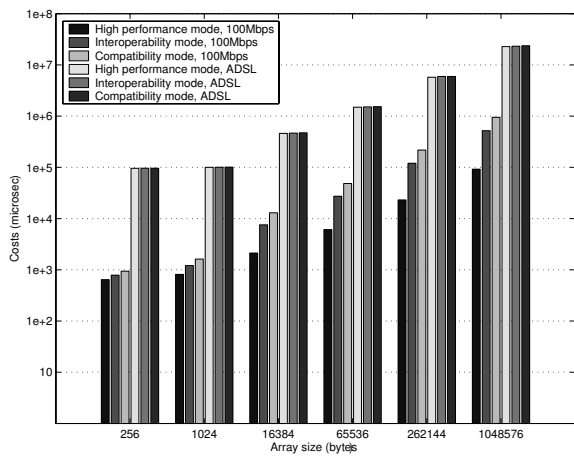


Figure 4. Comparison of overall costs in high performance, interoperable and compatible modes over 100Mbps and ADSL links for integer arrays.

a minimum. XML compression will increase the computation on the server side as the number of clients increases, whereas with PBIO, the server simply sends data in its native format.

Comparison among the Modes of Operation. Based on these measurements, it is possible to characterize the performance of each of the three modes of SOAP-bin operation discussed in Section 1.

Figure 4 shows the costs incurred in the three modes of operation. It is evident that, for high bandwidth links, the differences in performance increase as large data sizes are involved, whereas the costs over low bandwidth links are similar. This is because of the large delay introduced by slow links, which overshadows any smaller delays due to XML conversion at either end. This leads us to conclude that the high performance mode of operation should be used in "internal" communications between back-end servers, whereas the interoperable or compatible modes of operation should be used if the data is required to be in, or available as XML format.

Continuous Quality Management. In all experiments, RTT is measured during each request using the same method used in TCP implementations, with future work planning to use more effective estimators. Specifically, in the current implementation, a client sends a timestamp to the server along with the message, and the server sends back the same timestamp along with the reply. The client then computes the difference to determine the RTT for that request. This RTT value is used to update the client's measure of the cumulative RTT value through exponential averaging. While calculating RTT, the server can exclude the cost of prepar-

ing the response, by setting the timestamp forward by that time.

The information given in the quality file is used by both the client and the server just before sending the message. Based on the estimated RTT value, the corresponding interval in the policy is selected and the appropriate message type is chosen for transmission. The current message is then copied onto the chosen type through a single copy and then sent for marshalling. Every time the RTT is estimated by the client, the server is informed of the new value during the next request.

4.3. SOAP-binQ: Sample Applications

We present three applications - an image application, a commercial application and a scientific visualization application that are adapted to use SOAP-binQ.

4.3.1. Image Application We have created a real-time imaging code similar in structure to the Skyserver [19] application. The setup consists of a collection of servers, each of them possessing a set of images collected by remote telescopes. One of these servers acts as the primary server, in that it is directly accessible by a web user. Requests for images are directed at the web server, which then reroutes the request to an appropriate server among the collection of image servers.

For the image server application, due to its wide area nature, we evaluate its behavior in response to changes in network conditions. The server and client are run on two PCs connected by a single-hop 100Mbps Ethernet link, each with a 2.2GHz Pentium IV processor and 512MB RAM, running Linux 2.4.18-27.7. The application starts with the client sending a request to the server for an image, identified by its filename, and an operation to be performed on it. In this case, it is edge detection on PPM images (Skyserver uses JPEG images, for which we are currently implementing suitable data transformations). The PPM images are 640x480 pixels in resolution, with 3 bytes per pixel, one for each color value. Hence, the ideal response is close to 1MB in size. This results in high response times, especially for slow network connections.

For this application, the quality file is written to allow the server to resize the output image to 320x240 resolution when response times are high. In practice, more than two image sizes and/or additional ways of reducing response data would be used, but these experiments use only two image sizes, since the intent is simply to demonstrate the utility of this functionality offered by SOAP-binQ. To emulate network variations, cross-traffic is introduced using the IPerf tool *iperf*, which sends UDP packets at varying speeds. The resulting response times are shown in Figure 5.

As evident from the measurements in Figure 5, runtime quality management enables the application to send high-

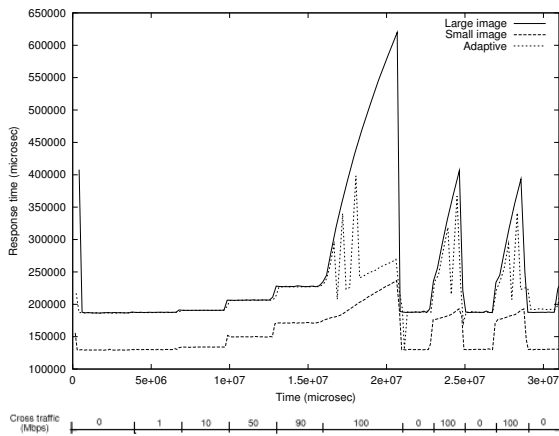


Figure 5. Response times for Image Application.

resolution images in good conditions, but once the response time increases further than that specified in the policy, it changes to sending lower resolution images. When conditions improve, it reverts to the original image sizes. As a result, the adaptative method’s performance lies ‘between’ the performance attained for large vs. small image files.

4.3.2. Commercial Application Our next application emulates the operational information systems used by airlines, transport companies (e.g, FedEx), and others. In this application, information is continuously produced, entered in a large, memory-resident data set, business rules applied to it, and resultant data is shared with end users. In the specific scenario used here, flight and passenger information is collected and distributed, and excerpts of such information are shared with relevant parties, such as flight caterers. The client, in this case, requests specific detail about the meals to be served, and the server responds with such detail.

Experiments assume an operational information system that uses PBIO in its core communications, but must convert catering information to XML for interoperability purposes. Measurements compare the transport of XML vs. PBIO data to end users, using the ADSL link. As seen in Table 1, the smaller data sizes used by PBIO result in improvements compared to the direct use of XML, requiring a PBIO ‘plug-in’ at the client side. Improvements would be more substantial if conversion to XML was not performed at all, directly transmitting selected PBIO data from the core system infrastructure to plug-in capable end users. This in fact, is the way in which PBIO-based communications are used by one of our collaborators, Delta Technologies[8].

4.3.3. Scientific Visualization Application Collaborative scientific applications can generate and consume substan-

	Size	Event rate (events/sec)
SOAP	3898 bytes	10.15
SOAP-bin	860 bytes	13.76
Native PBIO	860 bytes	14.06
SOAP (compressed XML)	1264 bytes	13.17

Table 1. Event rates for airline application

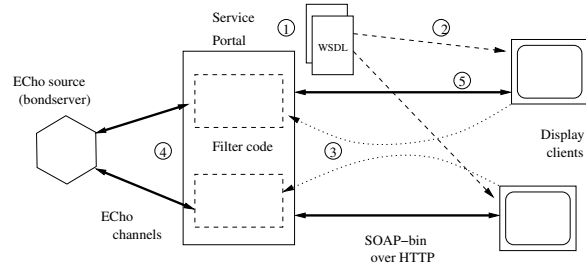


Figure 6. Remote Visualization Application - Architecture.

tial amounts of data [13]. The example used in this paper is from the domain of molecular dynamics, where the application models the behavior of the bonds between atoms within a molecule over time. It consists of a “bond server” that constructs a graph, where the vertices represent the atoms and the edges represent bonds. This data is available for a sequence of timesteps. Such a graph is constructed for every timestep and sent to a remote client for processing/display. The database on the server’s end stores data in the raw format, and the display expects data in SVG format. The overall architecture of this framework is as shown in Figure 6.

The display client is connected to the service portal through a HTTP connection. The service portal acts as a sink for the ‘Echo’¹ event source that generates bond data. The portal thus has an ‘ECHO’ bondserver as a backend. The service portal (1) advertises its services through a set of WSDL files. These are obtained by the display clients (2), which then construct the appropriate request (3), with filter code and the desired output format. Data arriving from the bondserver (4) is then modified by the filter code, providing the output in the desired format, which is then sent back to the client (5) as the response. The client can dynamically change the filter code and the output format desired. This has been successfully implemented, providing more flexibility to the client in obtaining the data it needs. Measurements conducted over the same setup as that used

¹ Echo is our own implementation of a publish/subscribe, event-based communication system, focused on large-data applications[5].

in image application shows a response time of about $2400\mu s$ for a data size of 16Kbytes, indicating a response time low enough for visualization purposes.

5. Conclusions and Future Work

This paper describes the software architecture for an efficient realization of the SOAP protocol. The goal is to make SOAP more broadly useful for end user applications, particularly targeting large-data applications [13]. The key idea is to use SOAP's XML-based parameter data as meta-information, while actual data exchanges between clients and servers utilize binary representations of such data. The SOAP-bin protocol presented in this paper has substantial performance advantages compared to SOAP, due to the reduced data sizes for binary vs. XML data and due to reductions in processing overheads for these two alternative data representations.

A generalization of SOAP-bin, termed SOAP-binQ, further extends this protocol by associating runtime quality management functions with SOAP parameters. The idea is to use application-specific data manipulations, such as data downsampling, to adjust data volumes to available clients. The adaptive behavior implemented by SOAP-binQ further expands the range of applications that can operate with the SOAP protocol. Continuous quality management is particularly important in resource-constrained environments, like international Internet connections, for instance.

Currently, Soap-binQ quality handlers manipulate only binary data. In future work, we will generalize handlers to be able to manipulate XML data, binary data, or both. In addition, our current implementation installs handlers statically, at compile-time. In other work [5], we have already developed the technologies necessary to install binary handlers at runtime, using dynamic binary code generation techniques and/or using code repositories. We are also pursuing the use of SOAP-binQ with more complex end user applications, the near-term goal being its use in the Smart-Pointer application[21]. The intent is to leverage SOAP's promise of interoperability for high end codes.

References

- [1] G. Alonso. Myths around web services. *Bulletin of the Technical Committee on Data Engineering*, December 2002.
- [2] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In *Proceedings of The Eleventh International Symposium on High Performance Distributed Computing*, 2002.
- [3] J. Clark. Expat - XML parser toolkit. <http://www.jclark.com/xml/expat.html>.
- [4] H. Dai, S. Mishra, and M. Hiltunen. Corba-as-needed: A technique to construct high performance corba applications. In *Proceedings of the 9th IEEE International Conference on High Performance Computing*, December 2002.
- [5] G. Eisenhauer. The ECho event delivery system. <http://www.cc.gatech.edu/systems/projects/ECho/>.
- [6] G. Eisenhauer, F. E. Bustamante, and K. Schwan. Native data representation: An efficient wire format for high performance computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(12), Dec. 2002.
- [7] G. Eisenhauer, F. Bustamante, and K. Schwan. A middleware toolkit for client-initiated service specialization. In *Proceedings of the PODC Middleware Symposium*, July 2000.
- [8] A. Gavrilovska, K. Schwan, and V. Oleson. A practical approach for 'zero' downtime in an operational information system. In *International Conference on Distributed Computing Systems*, July 2002.
- [9] A. Graveley. Making SOAP with SOUP. <http://lwn.net/2001/features/OLS/pdf/pdf/soup.pdf>.
- [10] Q. He and K. Schwan. IQ-RUDP: Coordinating application adaptation with network transport. In *Proceedings of the 11th International Symposium on High Performance Distributed Computing*, 2002.
- [11] J. Keeney. Chisel: A policy-driven, context-aware, dynamic adaptation framework. In *Proceedings of the Fourth IEEE International Workshop on Policies for Distributed Systems and Networks*, 2003.
- [12] Network for earthquake engineering simulation. <http://www.nees.org>.
- [13] B. Plale, V. Elling, G. Eisenhauer, K. Schwan, D. King, and V. Martin. Realizing distributed computational laboratories. *The International Journal of Parallel and Distributed Systems and Networks*, 2(3), 1999.
- [14] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha. On adaptive resource allocation for complex real-time application. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, December 1997.
- [15] R. Salz. Transporting binary data in SOAP. <http://webservices.xml.com/pub/a/ws/2002/08/28/endpoints.html>.
- [16] B. Seshasayee, K. Schwan, and P. Widener. SOAP-binQ: High performance SOAP with continuous quality management. Technical report, CERCS, GIT-CERCS-03-30.
- [17] S. Shirasuna, S. Matsuoka, H. Nakada, and S. Sekiguchi. Evaluating web services based implementations of GridRPC. In *Proceedings of the 11th International Symposium on High Performance Distributed Computing*, 2002.
- [18] Simple object access protocol SOAP 1.1. <http://www.w3.org/TR/SOAP/>.
- [19] A. Szalay, J. Gray, A. Thakar, P. Z. Kunszt, T. Malik, J. Rad-dick, C. Stoughton, and J. vandenBerg. The SDSS sky-server - public access to the sloan digital sky server data. <http://skyserver.sdss.org/en/sdss/skyserver/>.
- [20] P. Widener, G. Eisenhauer, K. Schwan, and F. E. Bustamante. Open metadata formats: Efficient XML-based communication for high-performance computing. *Cluster Computing: The Journal of Networks, Software Tools, and Applications*, (5):315-324, 2002.
- [21] M. Wolf, Z. Cai, W. Huang, and K. Schwan. Smartpointers: Personalized scientific data portals in your hand. In *Proceedings of SuperComputing 2002*, November 2002.