

Building a Generic SOAP Framework over Binary XML

Wei Lu¹, Kenneth Chiu², Dennis Gannon¹

1. Computer Science Department, Indiana University

2. Department of Computer Science, State University of New York (SUNY) at Binghamton

Abstract

The prevailing binding of SOAP to HTTP specifies that SOAP messages be encoded as an XML 1.0 document which is then sent between client and server. XML processing however can be slow and memory intensive, especially for scientific data, and consequently SOAP has been regarded as an inappropriate protocol for scientific data. Efficiency considerations thus lead to the prevailing practice of separating data from the SOAP control channel. Instead, it is stored in specialized binary formats and transmitted either via attachments or indirectly via a file sharing mechanism, such as GridFTP or HTTP. This separation invariably complicates development due to the multiple libraries and type systems to be handled; furthermore it suffers from performance issues, especially when handling small binary data. As an alternative solution, binary XML provides a highly efficient encoding scheme for binary data in the XML and SOAP messages, and with it we can gain high performance as well as unifying the development environment without unduly impacting the web service protocol stack. In this paper we present our implementation of a generic SOAP engine that supports both textual XML and binary XML as the encoding scheme of the message. We also present our binary XML data model and encoding scheme. Our experiments show that for scientific applications binary XML together with the generic SOAP implementation not only ease development, but also provide better performance and are more widely applicable than the commonly used separated schemes.

1 Introduction

As the primary technology for Service Oriented Architectures (SOA), web services are becoming the most widely accepted and successful type of services for building large-scale distributed systems. The success of web services technology can be largely attributed to XML, which provides a unique but crucial feature for the success: interoperability. XML-based concepts and abstractions are inextricably

intertwined with web services. As a result, the entire protocol stack of the web service oriented architecture, from the fundamental WSDL to the state-of-the-art WS-* set of specifications, heavily rely on the XML model and related operations, such as XSLT, XQuery, and so forth.

SOAP [21], as a messaging protocol based on the XML data model, has been the dominant means of communication between web services. Though not strictly required by SOAP, XML 1.0 [24] is its default encoding scheme, and therefore the *de facto* standard wire format for web services. However it is well known that the processing of the SOAP message encoded in XML is usually slow and memory intensive, due to the verbose syntax of XML and the textual nature of its serialization [9, 17, 10].

For scientific applications the most common data types are numeric data and vectors of such data, and our earlier work [9] has concluded that for them the conversion between the native floating-point number to their textual ones dominates the SOAP performance. In a number of scientific applications, such as distributed data mining [14], a large binary data set usually must be transmitted; while for the other ones, such as wide-scale wireless sensor networks, small data messages are transmitted between the machines but at very high frequency and on real-time demand. For either case the processing overhead of SOAP messages is intolerable.

Thus the conventional usage of web services in scientific applications is that web services are for *control*, while *data* must be stored, managed, and transmitted using binary, non-XML-based formats. To maintain this separation between control and data, distributed scientific applications such as LEAD [1] typically handle data in a non-web-services-based, ancillary framework consisting of binary data formats, various mechanisms from transmitting files in these format, and the additional libraries and APIs for using these formats. For example, the *data* can be saved as a netCDF [18] file which is accessible via HTTP, GridFTP[3] or other file sharing mechanisms, and a SOAP message containing a link to the file is sent through the *control* channel to indicate the application where to pull the data from; alternatively the *data* in the base64 format is pushed

to the application side within the same channel of *control* but as an attachment via the various attachment facilities (e.g., WS-Attachment[19]). No matter which mechanisms we are using, the result is essentially two systems that parallel each other: a web service infrastructure for the control of scientific application, and a bespoke infrastructure for the scientific data.

Treating data in a completely different conceptual framework has several disadvantages. Beyond the cost of using and maintaining a separate code infrastructure for the data, users and developers must bear the additional mental cost of learning two different sets of concepts, terminology, and software. Furthermore, the separated solution does not necessarily produce the improvement of the system performance as expected. As we will show later in some cases, especially when the binary data set is relatively small, it actually even has worse performance than the general SOAP over textual XML. Last but not the least, with the growth of SOAs the web service protocol stack is becoming deeper and wider, so if the data was treated in a non-XML-based formats the entire stack would be fundamentally impacted, even perhaps becoming totally incompatible.

We posit that the perceived performance problems of XML are not due to its logical structure, but rather to its lexical and syntactic format. This suggests addressing XML's performance issue by developing an alternative, efficient serialization of this logical structure. With these alternative serializations, scientific data could be directly transmitted as "binary XML", thus directly integrating data into the web services framework, and obviating any special treatment. The resulting unified framework will bring the benefits of web services to the data as well as the control of scientific applications. A common API representing the logical structure can be used for both textual XML and binary XML, thus allowing applications or the higher layers of web service stack to transparently use either serialization without code modification.

In previous work, we presented a binary XML for scientific applications (BXSA) [8], and a set of encoding rules in the context of the XML Infoset model. We also showed that BXSA could be fast compared against scientific data formats such as netCDF. In this work, we further support the suitability of the binary XML for scientific applications, by developing a scientific-data-friendly XML data model (bXDM) as well as the encoding rules for it. Also to investigate the benefit of binary XML for web services and SOAs, we implement a generic SOAP implementation which supports both textual XML and binary XML as the encoding scheme of the message. Following the generic programming paradigm, the generic SOAP engine has been proven to be very flexible and extensible in handling the complexity of the lower-level encoding, transport binding and their interactions with performance. Because the binding is at

compile time, compiler optimizations are not impacted, and inlining is still enabled.

In this paper, we present the structure and implementation of above mentioned modules and performance experiments which show that binary XML together with generic SOAP implementation challenges the conventional wisdom that scientific data must necessarily be divorced from web service concepts and principles. Although we understand that change must be gradual, we believe that eventual unification will lead to benefits, both foreseen and unforeseen.

The rest of the paper is organized as follows. Section 2 introduces the background of the SOAP; Section 3 and Section 4 describe the structures of bXDM and BXSA; The generic SOAP implementation is introduced in the Section 5; We present in Section 6 performance results; Related work is discussed in Section 7.

2 SOAP

SOAP is a simple messaging protocol designed for web services to exchange information in a distributed and heterogeneous environment. It supports multiple message exchange patterns, including one-way messaging and various request-response type exchanges.

To achieve high interoperability, the SOAP message is formally specified as an XML Infoset, which provides an abstract description of the content. Naturally XML 1.0 is the *de facto* "wire format" of the SOAP message and HTTP is usually adopted to carry the message due to its ubiquitousness. However, SOAP does not mandate this binding combination. Actually it intentionally leaves the message encoding and transport protocol open so as to be independent of the lower level binding layers. Other alternative representations (e.g., compressed or binary ones) and transport protocols (e.g., SMTP or TCP) can be used if appropriate. Users are free to specify the alternative message encoding/binding scheme in the WSDL file, though most implementations support this flexibility either poorly or not at all. Based on the SOAP layer, then the other web service protocols in the protocol stack, ranging from the security (WS-Security [15]) to workflow (BPEL4WS [16]), are built up (as shown in Figure 1). Separated by the SOAP layer theoretically they should be unaware of the underlying encoding and transportation layers.

3 Extended XDM

XML is designed as a textual format for semi-structured data. In order to represent the logical structure of the XML document several specifications have been defined, such as XML Infoset [23], Post-Schema Validation Infoset and XQuery and XPath Data Model (XDM) [25]. Those data

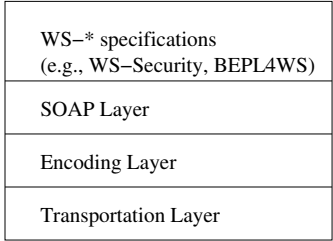


Figure 1. The web service protocol stack.

models essentially serve as an abstraction layer for the information which is stored in the XML documents. The actual application layer is thus protected from the lower-level serialization details. All fundamental building blocks of web-services-oriented architectures, such as XPath, XSLT, SOAP and so on, rely on this abstraction layer to describe the data they are processing.

We selected XDM rather than the more well-known XML Infoset as the core XML data model in our implementation, because beyond the information contained in the XML Infoset, XDM also supports typed atomic values and XML Schema type information in the data model itself. The type information makes XDM promising, since type is a very important hint for coding efficiency. For instance the typed atomic value allows our API to represent numbers in their native machine form rather than as a character string as required in the XML Infoset, therefore obviating the expensive conversion between machine form and ASCII. Based on our experience this small improvement indeed is the key step toward the high performance encoding of scientific data in XML. However XDM is still awkward for handling the numeric-data intensive cases in scientific application, especially the representation of array for which XDM needs to generate numerous nodes repeatedly for the items in the array. Therefore we extended XDM to efficiently handle arrays of numbers, which we call *bXDM*.

bXDM includes the seven node types defined in XDM (Document, Element, Attribute, Namespace, PI, Text, and Comment), but also further refines the Element node into two subtypes: LeafElement and ArrayElement. The LeafElement represents the element with primitive atomic data type, such as integer and double. We implement the LeafElement by using templates in C++, and it is declared as `LeafElement<T>` where T can be any primitive data type. So the atomic data value of the LeafElement can be saved as the machine native format without any overhead. The ArrayElement is designed to represent an array of same atomic typed items as a single element in the model. Similarly the ArrayElement is declared as `ArrayElement<T>` where T is any primitive data type. Internally the data value of the ArrayElement is stored as a packed array of primitive types, which is

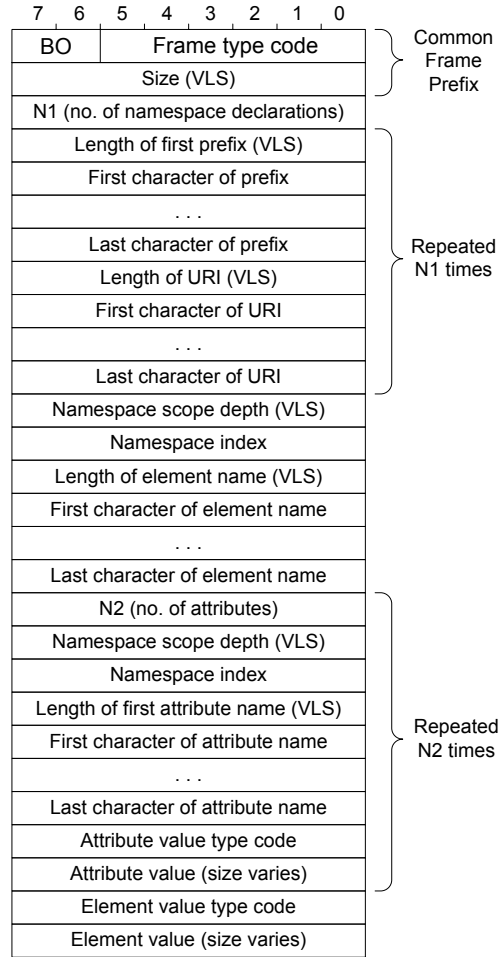


Figure 2. The basic structure of the BXSA frame.

compatible with most programming languages used in high-performance computing, such as C and Fortran. This allows the ArrayElement to efficiently represent arrays and matrices as well as raw octet streams.

4 BXSA

To show a proof-of-concept that binary XML is a viable approach to unifying control and data for web services, we developed our own binary XML encoding scheme, called BXSA (Binary XML for Scientific Applications), particularly for the domain of scientific computing. BXSA is a layered format, and depends on XBS [7] to pack fundamental types into a sequence of bytes. The XBS format is a minimalistic format that supports 1-, 2-, 4- and 8-bytes integers; 4- and 8- bytes floating -point numbers, and 1-dimensional array. All numbers are aligned to a multiple of each types size. Both little-endian and big-endian for-

mat are supported. Based on XBS, BXSA defines a set of encoding rules for the bXDM data model.

4.1 Frames

A BXSA document consists of a sequence of *frames*, each of which will represent a corresponding node in the bXDM model. One *frame* can contain other child frames as its content model, and thus the tree structure in the bXDM is represented by this recursive embedding relationship between frames.

The basic layout of the frame is shown in the Figure2. Every frame has a Common Frame Prefix, whose size is at least 2 bytes. The first two bits in the Common Frame Prefix, named byte-order, is used to indicate the endianness of data in the frame. Associating the byte-order bits with each frame rather than the entire BXSA document makes it simpler to embed the frame within other documents without regard to a possible different byte order used by the container.

The following 6 bits in the Common Frame Prefix are used to identify the kinds of the frame, which are mapped to the node types in bXDM. The implementation would be easy if we created a corresponding frame type for each node type in the bXDM model. However it will lead to numerous, small frames for the attributes and namespace declarations in the BXSA document, hence degrading the encoding efficiency. So we enlarge the granularity of the frame to be coarse enough to represent all types of Element nodes in the bXDM model and have the attributes and namespace declarations be the part of the frame. The *Size* field in the frame indicates the number of bytes of the frame in a variable-length integer format, and it enables the accelerated sequential access ability, by which we can sequentially scan frames without fully parsing all parts of the document.

After the Common Frame Prefix follows the content model of the frame, which varies with different frame types.

Leaf Element Frame: This frame is used to code the LeafElement in the bXDM. Following the common frame prefix, a Leaf Element Frame can contain the namespace declarations, the element name, the attributes, and finally the element value itself.

Each element frame contains a namespace symbol table, whose entry consists of prefix and the URI of a namespace declaration. In textual XML the namespace is referred by its prefix by any QName (qualified name) in its scope. However the prefix is inefficient and unnecessary for binary XML representation. To save the space, BXSA uses tokenization for the namespace reference, which means the namespaces in any QName are referred to via its index in the namespace symbol table of the frame. Since an element may use a namespace declaration created by its parent, a namespace

reference also includes the namespace scope depth. This is a count backwards to indicate where the namespace was declared.

Since LeafElement has typed value, the element value in the frame also is typed, which means it is stored in the native machine format in the frame with an auxiliary type code.

Component Element Frame: This frame is used to encode a general element that has child elements. Its format is similar to that of a Leaf Element Frame, except that rather than a type code followed by the value, there is a count followed by a sequence of frames. Each frame defines either character data or a child element.

Array Element Frame: This frame is used to encode the ArrayElement of bXDM data model. The format of the frame is identical to that of a leaf element, except that rather than being following by a single number, it is followed by a count and a packed sequence of numbers in their native formats. Since the value of the ArrayElement in the bXDM model is an aligned, packed array, large arrays can be read or written by simply using memory-mapped file I/O. This will avoid an extra copy, making such I/O efficient.

Character Data Frame, PI Frame and Comment Frame: The Character Data frame is used to encode character data in mixed content compound element frames. Its format consists of the common frame prefix followed by a count of characters, followed by the sequence of characters. The PI framework and Comment frame are self-explanatory and have the exactly same structure with the Characters Data frame except the different frame type code.

4.2 Transcodability between BXSA and XML

A binary format that is transcodable to XML can be converted to textual XML, and then back to binary XML without change. Such a binary format must also support the ability to convert a textual XML document to binary XML, and then back to the textual format without change. This is important for scientific applications for interoperability purpose. Some required tools may only work with textual XML. The primary challenge for supporting this property is the handling of numeric data. BXSA currently transcodes except for the handling of floating-point numbers, which are converted to full precision regardless of the original input. Also if the schema of the document is unavailable, the XML serialization of bXDM should contain the type information explicitly, as required by the SOAP encoding rule [22], otherwise we are not able to create the typed LeafElement in the bXDM model.

5 Generic SOAP implementation

As mentioned earlier, SOAP allows the diversity of message encoding and transportation schemes which, however, is a challenge for SOAP implementations. Ideally the implementation should be general enough to be independent of the lower level encoding and binding schemes, and it also should be extensible to allow new customized encoding and binding schemes to be added. Furthermore the implementation should be able to cope with the combination of those various encoding, binding schemes as well as other web service features (e.g., security). For instance, a desired SOAP implementation should enable a web service to use XML 1.0 encoding over the SMTP with the XML signature applied, while allowing another web service to use the binary XML encoding over the HTTP without any security consideration.

To address this sort of highly customizable design and combinatorial behavior, the generic programming [5] and template technology of C++ usually are adopted as they generate code at compile time based on the types provided by the user. The design paradigm is the so-called policy based design [2], in which each template type is considered as one of the policies the template class will be enforced with. Every policy is just defined as an abstract concept with a set of *valid expressions*. For example our generic SOAP engine is declared to have two template parameters, *EncodingPolicy* and *BindingPolicy*, which are responsible for the encoding and binding scheme the engine will adopt.

```
template <class EncodingPolicy,
         class BindingPolicy>
class SoapEngine
{...};
```

It will be straightforward to introduce more policies (e.g., a security policy) into the generic engine by just adding more template parameters.

Any class can be used by the generic SOAP engine as long as it conforms to all the *valid expressions* of the policy concept. The class usually is called a *model* of the policy concept and it stands for a concrete implementation of that policy. By assigning different classes to the template parameter, a user can easily handle the combinatorial problem of the encoding/binding scheme. For example

```
SoapEngine<XMLEncoding, HttpBinding> soapXML;
```

defines a SOAP engine which uses XML 1.0 as the encoding scheme and HTTP as underlying transport protocol, while

```
SoapEngine<BxsaEncoding, TcpBinding> soapBin;
```

defines a SOAP engine which uses BXSA as the encoding scheme and raw TCP as underlying transport protocol. Obviously we can have two more combinations of encoding/binding schemes and all of them can be used by the generic engine in a type-safe and highly efficient way.

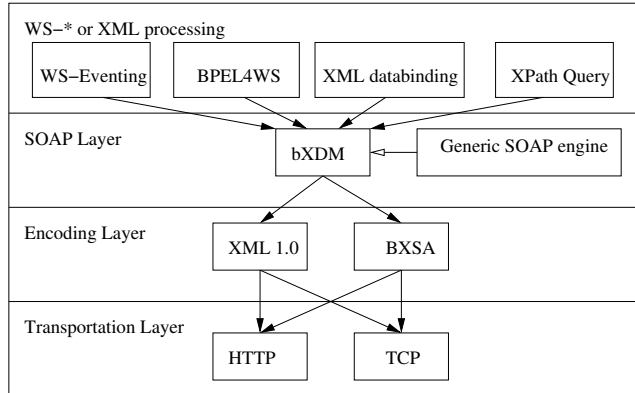


Figure 3. The architecture of the WS stack based on the generic SOAP implementation.

5.1 Architecture

Our generic SOAP library implements the basic models defined in SOAP specification, while the user is responsible for providing the concrete implementations for the encoding and binding policy at compile time.

In the generic SOAP engine, the SOAP message is modeled in the bXDM model instead of the XML Infoset. So to send a SOAP message, the generic engine first constructs the SOAP message in the bXDM model, then invokes the encoding policy provider to serialize the message into the octet stream, finally transferring the stream by calling the binding policy provider. Receiving the message is just the reverse procedure.

The conceptual architecture of the system is shown in the Figure 3. Those layers (e.g., WS-eventing, WS-addressing and so on) above SOAP are bXDM oriented, and thus are ignorant of the underlying encoding and transport layers. And since bXDM is extended from XDM, any XDM-based XML processing (e.g., XPath or XSLT) should be able to run with binary XML with minor modification.

The generic architecture simplifies the development of the SOAP-based server, particularly that of intermediary SOAP nodes. SOAP messages are designed to be transferred in a hop-by-hop style between the SOAP nodes and the bindings between the hops can be various and depend upon the intermediary node. Aided by the generic SOAP library, the intermediary node can just simply deploy multiple generic SOAP engines with different policy configurations to serve the up-link and down-link message flows. Furthermore, transcodability enables BXSA to be the intermediate protocol over the message hops, even when the message sender and receiver are communicating via textual XML.

5.2 Encoding policy

The encoding policy is an object that is able to serialize and deserialize the bXDM model. To separate the bXDM data model from the various encoding schemes we adopt the *Visitor* design pattern [12], in which every encoder behaves as a generic visitor of the bXDM data model and generates the specific serialization during the visiting. Therefore, the encoding policy first is a refinement of the *Visitor* concept of the bXDM model. Also for the deserialization, the encoding policy should contain a *factory method* [12], which parses the specific encoded stream and create the corresponding bXDM model.

At present our system provides two default implementations of the encoding policy, XMLEncoding and BXSAEncoding, for XML 1.0 and BXSA encoding respectively.

5.3 Binding Policy

The binding concept is an object that implements the binding rules for carrying a SOAP message within or on top of another protocol (underlying protocol) for the purpose of exchange. It simply provides the below self-explanatory, valid expressions to support the SOAP message exchange patterns:

- `int send_request(SoapEnvelope&);`
- `SoapEnvelope receive_response();`
- `SoapEnvelope receive_request();`
- `int send_response(SoapEnvelope&);`

How to implement the interface depends upon the specific transport protocols. For example to send the SOAP request the HTTP binding will create a HTTP request message with the serialized SOAP message as payload, while the TCP binding will just dump the serialization directly to a TCP connection. Currently the `HttpBinding` and `TCPBinding` are supported as defaults.

6 Experimental Studies

To investigate the benefit of binary XML to scientific applications, we performed experiments to compare the SOAP over BXSA binding against the conventional separated usage of web service for scientific applications as described in Section 1. Just like the typical configuration in the conventional separated usage, we use netCDF as the serialization format for the binary data due to its high encoding efficiency and popularity among scientific applications; meanwhile we

adopt the widely-used GridFTP or HTTP as the data transport channel in the separated scheme.¹ The details of the test programs are elaborated as below.

1. **Unified solution:** The client constructs the binary data in the bXDM model, then sends both the request and the binary data in one SOAP request message to the server. The SOAP message can be bound with BXSA/TCP or XML/HTTP. Once the server receives the message, it deserializes it into the bXDM model, verifies each value in the model, and sends the verification result back to the client in the SOAP response message.
2. **Separated solution:** The client first generates the binary data and saves it into a netCDF file, which will be retrieved by the server via HTTP or GridFTP; then the client sends the request in a general SOAP request message, whose content is just the URL pointing to the netCDF file, to the server, which in turn downloads the netCDF file, reads and verifies the file and finally sends the verification result back to the client in the SOAP response message.

The binary data model we are using in the experiments was derived from a sample file used for LEAD project, and consists of atmospheric information, which depends on four parameters, namely time, y, x and height. Basically the data set consists of two equal-size arrays:

- an array of 4-byte integers as the index and
- an array of double-precision, 8-byte floating point numbers to represent the dimension values.

For convenience of notation we call the length of the array the *model size* of the data set.

Our experiments were performed over both local and wide area networks, and the response time at the client side was measured.

6.1 Size of Serialization

First, we tested the size of the serialization result of the numeric data by BXSA, netCDF and XML 1.0 respectively. For 1000 pairs of float-pointing and integer numbers (i.e., the *model size* = 1000), the native representation requires at least 12K (i.e., $1000 * (4 + 8)$) bytes. The result in Table 1 shows that both BXSA and netCDF have a insignificant encoding overhead, which are 1% and 2% respectively. However, XML encoding introduces 99% encoding overhead even if it is namespace free and uses the shortest `<i>`

¹We skip the tests of the attachment solution, since it is not widely-adopted by the scientific applications and furthermore in terms of performance it should be close to SOAP with HTTP data channel solution.

Format	Size (bytes)	Overhead
Native representation	12000	0%
BXSA	12156	1.3%
netCDF	12268	2.2%
XML 1.0	23896	99.1%

Table 1. Serialization size of the binary data set with *model size* = 1000.

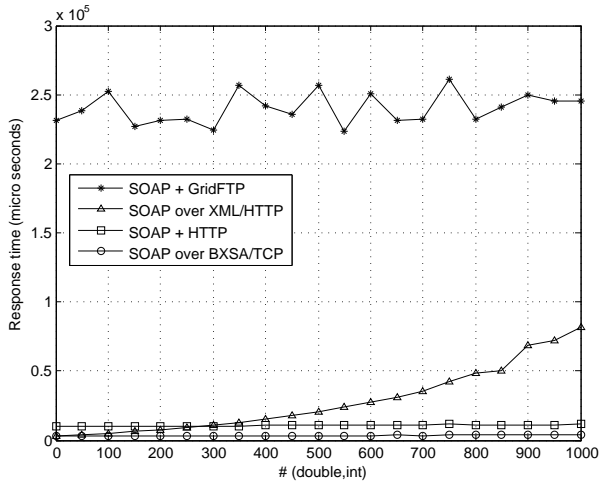


Figure 4. Message response time when running with small binary data set.

as the tag name of each element in the array. Moreover the overhead of XML encoding is linearly proportional to the *model size*. Apparently the open/close tag pair per element required by XML 1.0 contributes to the significant encoding overhead.

6.2 Message Response Performance on LAN and WAN

The performance test in the local area is performed on the LAN with a 0.2 milliseconds round trip time (RTT). The client program and server program are running on two Linux 2.6.11 boxes, both with a 2.8 GHz Pentium CPU and 1GB of RAM. As mentioned previously, the server program also behaves as the client of GridFTP or HTTP to download the netCDF file. The GridFTP C client library and libcurl are used as the client library of GridFTP and HTTP, while the machine running the client program hosts GT4 GridFTP server and the Apache web server v2.0.

We first measured the message response time for small-sized messages in which the *model size* of the data set ranges from 0 to 1000. The result is presented in Figure 4. We see that SOAP over BXSA/TCP achieves superior per-

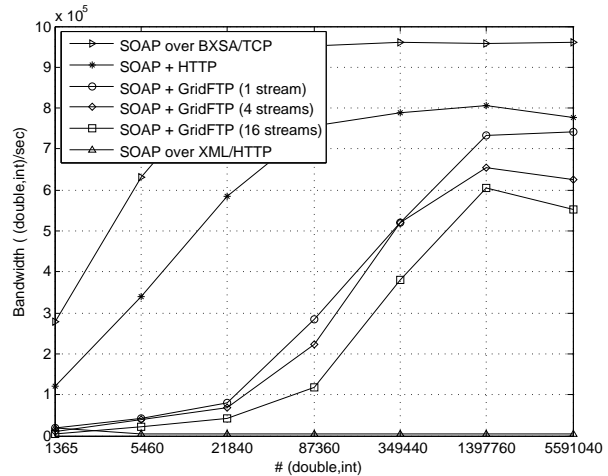


Figure 5. Invocation performance when running with larger binary data over LAN.

formance over other schemes. The standard SOAP over XML/HTTP binding performs well when the message is fairly small, but as the size of the message increases linearly, its response time increases exponentially, and is even more expensive than the separated solution, namely SOAP with HTTP data channel. This confirms our earlier observation [9], that the performance bottleneck is not merely the size of the serialization, but actually lies at the conversion between floating-point numbers and their ASCII representation. Both the BXSA and netCDF encode the numeric value natively, thus eliminating the need for conversion, so the two schemes, SOAP over BXSA/TCP binding and SOAP with HTTP data channel, present a flat linear increase rate. The difference between the two schemes is predictable as the SOAP with HTTP data channel needs two separated communication channels and extra disk I/O. It is possible to eliminate the extra disk I/O by tuning the HTTP library to process the data directly in memory. However, the netCDF library does not support reading the data directly from memory. The high response time by the SOAP with GridFTP data channel scheme is due to the expensive authentication and the SSL handshake protocol. This suggests GridFTP is unsuitable for the small message cases.

We next looked at the performance of those schemes when running with large messages. The *model size* of the data set in the message increased from 1365 to 5591040 exponentially. The numbers are selected so that the corresponding BXSA serialization size is from 16K bytes to 64M bytes. To make the result easily distinguishable, instead of the response time we measure the bandwidth which equals the *model size* divided by the response time. The experimental result is presented in Figure 5. We can see that the SOAP over BXSA/TCP scheme still shows the best per-

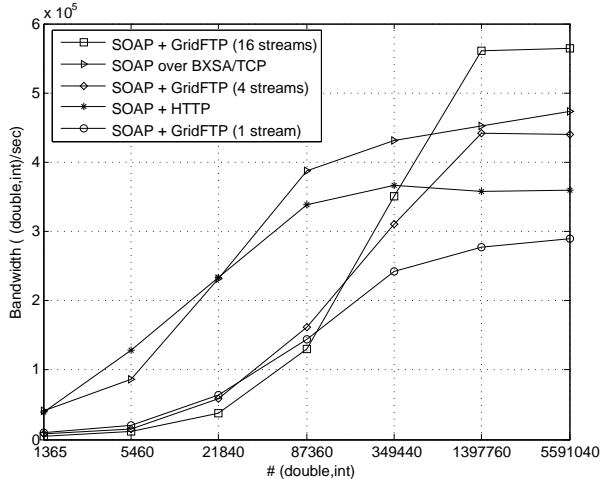


Figure 6. Invocation performance when running with larger binary data on WAN.

formance in this experiment. Its bandwidth curve is saturated at 960K pairs of floating-points and integers per second, which is almost 10MB/sec and has almost reached the maximum transfer rate for a single untuned TCP stream. The SOAP with HTTP data channel is little bit slower, we still speculate this performance difference is due to the extra disk I/O enforced by the netCDF library. The SOAP with GridFTP data channel begins to match the above two schemes; the overhead of the security is amortized as the message size increases. GridFTP supports parallel TCP streams for transferring data, but our observation is that over a LAN the parallelism in GridFTP provides little additional benefit, and indeed somewhat degrades performance. Researchers [4] attribute this to more “seek” operations at the receiver for the blocks received out of order. Lastly it is not surprising that SOAP over XML/HTTP scheme lost the game at the very beginning of the test.

In our last reported experiment, we repeated the above experiment but over a wide area network. Our wide area testbed contains a remote server, which is a Linux 2.6.8 machine, with a AMD Athlon XP 3000+ CPU and 1GB of RAM, located at the University Chicago, and the aforementioned machines at Indiana University as the client. The software and library configuration is same as the one of the local area testbeds. The round trip time between the client and remote server is 5.75 milliseconds. The result illustrated in Figure 6 shows that the ordering has partially changed. The parallel transport of GridFTP begin to show its benefit, in which GridFTP transport is not restricted by the bandwidth of a single TCP stream, thus boosting the transfer rate. Both SOAP over BXS/TCP scheme and SOAP with HTTP data channel have similar performance. They are still restricted by the bandwidth of a single TCP

stream, so the transfer rate has been the primary performance bottleneck. With our generic framework, however, we can easily rebind the BXS transport to multiple TCP streams, thereby eliminating this restriction.

Based on the experiment result, we can conclude that to construct web services for scientific applications:

- Binary XML provides not only an unified development environment, but also the widest applicable area for scientific applications in term of performance; it can meet the performance/scalability requirement of both the applications with small message size but frequent access rate (e.g., sensor networks) and the ones with large binary data size (e.g., distributed data mining).
- In the separated schemes, HTTP is a good candidate for the data channel due to its simple structure and high interoperability; GridFTP is appealing when security is a primary concern or when on the WAN, the binary data volume to be transferred is large enough.

7 Related Work

Most closely related to our work presented in this paper is the FastInfoset proposed by Sun [20], which makes use of ASN.1 modules for describing and encoding the XML Infoset. Similar with the SOAP over BXS binding, FastWebService is proposed as a SOAP implementation bound on the FastInfoset. Compared with our work, FastInfoset uses a more sophisticated tokenization technology for name encoding; and it also supports direct binary data encoding by the octet information item, a counterpart of the ArrayElement defined in bXDM. However the primary difference is FastInfoset is based on XML Infoset Set, that means FastInfoset cannot utilize the type information for the efficiency of modeling and encoding.

Millau [13] is an efficient format for XML, especially in terms of tokenization. It does handle single-precision floating-point numbers in IEEE 754 representation, but does not have a facility for representing arrays of numbers. Millau is primarily targeted toward the business community, and an XML-RPC implementation is built upon it.

The Data Format Description Language (DFDL) [11], which is derived from BinX [6], is a language for describing data formats. DFDL describes all aspects of a data format, from byte-order to structure. By mapping binary formats to XML Schema, DFDL can be used to parse binary data into an XML-based data model. Being descriptive, however, DFDL does not prescribe how the data is actually stored. That is, DFDL *describes* binary formats, while BXS *is* a binary format. DFDL and BXS may have some use cases which overlap, but are essentially complementary.

SOAP+ [14] is a variation of SOAP designed for data mining applications, and it belongs to the control-data-

separated architecture, in which the control channel relies on the general SOAP protocol, while the data channel employs a specialized network protocol, called UDT, for large and distributed data sets. Differing from the pull-based separation scheme described in Section 6 the data channel of SOAP+ is push-based, so the extra request-response message exchange will be eliminated, but it may be at the cost of interoperability.

8 Conclusion

In this paper, we have described our experience of building SOAP and web services over binary XML. The performance tests suggest that XML and web services are not inappropriate for scientific applications as is usually assumed. Instead, with binary XML the data and control information of scientific applications can be effectively merged into one web services framework, and will outperform the prevailing practice of separating data from control in most cases. Beyond that, with the help of the generic SOAP implementation, binary XML also provides some advantages, such as the unified development and the intact web service protocol stack, which those common solutions are unable to achieve.

Acknowledgment

We would like to thank Thomas J. Hacker from UITS for his insightful suggestion on the parallel TCP, and thank XuiHai Zhang of University Chicago for kindly providing the remote testbeds.

References

- [1] Linked Environments for Atmospheric Discovery project web page. <http://lead.ou.edu/>, 2005.
- [2] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison Wesley, 2001.
- [3] W. Allcock. *GridFTP Protocol Specification(Global Grid Forum Recommendation GFD.20)*, 2003.
- [4] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster. The globus striped gridftp framework and server. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC'05)*, 2005.
- [5] M. H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison Wesley, 1998.
- [6] BinX project web page. <http://www.edikt.org/binx/index.htm>, 2005.
- [7] K. Chiu. XBS: A streaming binary serializer for high performance computing. In *Proceedings of the High Performance Computing Symposium 2004*, 2004.
- [8] K. Chiu, T. Devadithya, W. Lu, and A. Slominski. A binary xml for scientific applications. In *Proceedings of e-Science 2005*. IEEE, 2005.
- [9] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of soap performance for scientific computing. In *HPDC '02: Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)*, page 246. IEEE Computer Society, 2002.
- [10] D. Davis and M. Parashar. Latency performance of soap implementation. *IEEE Cluster And The Grid*, 2002.
- [11] Data Format Description Language project web page. <http://forge.gridforum.org/projects/dfdl-wg/>, 2004.
- [12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [13] M. Girardot and N. Sundaresan. Millau: an encoding format for efficient representation and exchange of xml over the web. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications networking*, pages 747–765. North-Holland Publishing Co., 2000.
- [14] R. Grossman and D. Hanley. Experimental studies scaling web services for data mining using open dmix: Preliminary results. In *KDD-2004 Workshop on Data Mining Standards, Services and Platforms (DM-SSP 04)*, 2004.
- [15] IBM. Web Services Security (WS-Security). <http://www-106.ibm.com/developerworks/webservices/library/ws-secure/>, 2002.
- [16] IBM. Business Process Execution Language for Web Services version 1.1. <http://www-128.ibm.com/developerworks/library/specification/ws-bpel/>, 2005.
- [17] C. Kohlhoff and R. Steele. Evaluating soap for high performance business applications: Real-time trading systems. *WWW*, 2003.
- [18] Unidata - NetCDF. <http://my.unidata.ucar.edu/content/software/netcdf/index.html>, 2005.
- [19] H. Nielsen, E. Christensen, and J. Farrell. Ws-attachments specification. Technical report, IBM, <http://www-106.ibm.com/developerworks/webservices/library/ws-attach.html>, 2005.
- [20] P. Sandoz, A. Triglia, and S. Pericas-Geertsen. Fast infoset. <http://java.sun.com/developer/technicalArticles/xml/fastinfoset/>, 2004.
- [21] W3C. Simple Object Access Protocol (SOAP) 1.1. <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>, 2000.
- [22] W3C. SOAP Version 1.2 Part 2: Adjuncts. <http://www.w3.org/TR/2003/REC-soap12-part2-20030624/>, 2003.
- [23] W3C. Xml information set (second edition). <http://www.w3.org/TR/xml-infoset/>, 2003.
- [24] W3C. Extensible Markup Language (XML) 1.0 (Third Edition). <http://www.w3.org/TR/2004/REC-xml-20040204/>, 2004.
- [25] W3C. Xquery 1.0 and xpath 2.0 data model (xdm). <http://www.w3.org/TR/2005/CR-xpath-datamodel-20051103/>, 2005.