Optimising Web Services Performance with Table Driven XML

Alex Ng
alexng@ics.mq.edu.au
Department of Computing, Macquarie University, North Ryde, NSW 2109, Australia

Abstract

The performance of the SOAP protocol has often been regarded relatively poor and requiring undue amounts of processor time, storage and bandwidth due to its use of text-based, metadata-laden XML encoding. There are many proposals available to tackle this perceived problem, however none of these coherently consider the different aspects of the problem: (1) message size; (2) message structure; (3) accessing individual elements; and (4) interoperability with other Web Services protocols. The technique proposed in this paper: Table Driven XML (TDXML) takes such a coherent view and offers more compact messages, simpler message structure and easier access to individual elements when compared to conventional SOAP. TDXML also enables keyed access to individual elements to enable single pass message parsing for improved object serialisation and deserialisation. Experimental studies show that TDXML can be incorporated easily into a commercially available SOAP implementation with 30% improved latency performance. Furthermore, this paper reports that TDXML can also be implemented with other transport mechanisms and provide even better performance (over 100%) than SOAP in regard to network bandwidth and latency.

1. Introduction

The performance of the SOAP protocol has received a significant amount of attention from the research community and from industry. Some of these studies concluded that SOAP is unsuitable for high performance real-time applications [23, 27]. It was observed that some of the studies were conducted either using proprietary SOAP implementations [23] or impractical test cases and scenarios [15, 16]. Gray’s [24] work highlights that the performance landscape changes with increases in message size and complexity. Making the scene more complicated are contradictory benchmark figures reported by major vendors [3, 4]. To resolve this confusion, prior to the development of an optimisation mechanism for Web Services, this research had conducted a series of investigations [28, 29] and gained first hand in-depth knowledge of the performance characteristics of SOAP in the context of commercial implementations. A summary of the findings is given in Section 2.

Recently, we have seen considerable activity in the area of SOAP performance optimisation. One of the major developments has been the ratification of the SOAP Message Transmission Optimisation Mechanism (MTOM) [25] standard. Major vendors also have offered their own solutions that can boost the performance of Web Services. An example is the XML Serialiser Generator (Sgen.exe) tool that comes with Microsoft’s .NET Framework version 2.0 [5] and lets a user generate precompiled code that can be used by Web Services clients to serialise transmitted information, as well as letting them use the classes in the System.IO.Compression namespace to read and write data with the gZip [17] compression algorithm. Sun Microsystems offer the Java Web Services Developer Pack (Java WSDP) 1.6 release which contains a proprietary Fast Infoset technology [34] that uses ASN.1-based [19] binary encodings to reduce transmission and processing times for XML messages. Using binary encoding can improve performance but at the same time poses the concerns [32] of multiple binary representations (big or little endian) and interoperability with existing standards (e.g. Infoset and XML compression).

Table Driven XML (TDXML) is the result of a series of detailed investigations into the different challenges faced by the Web Services protocol stacks. TDXML differs from all the other optimisation mechanisms because most of the proposed solutions tackle only one or two aspects of the problem while TDXML takes multiple aspects into consideration and makes use of a combination of techniques. Some the techniques are presented in Section 4 where the design of TDXML will be explained. One major highlight of TDXML is that the data is serialised into tabular format. Figure 1 is an example showing different representations when using XML and TDXML to represent the values of an ‘employees’ object containing the details of two people.
Conclusions and future work are discussed in Section 6. Document/Literal characteristics of TDXML compared to the ASP.NET evaluation results and comments on the performance techniques being used. Section 5 presents the provides an overview of TDXML and explains the and the anticipated benefits of this proposal. Section 4 research work, including the decisions that were made and the SOAP protocol. Section 3 explains the goals of this studied by introducing a 14Kbyte JPEG image to a exchanged SOAP messages without introducing other components in a system, the tested Web Service message was embedded in the form of a string object of TDXML with other transport mechanisms. The Microsoft .NET Remoting technology was also used to verify the interoperability of TDXML with other transport mechanisms. The impact of Base64 binary object encapsulation was studied by introducing a 14Kbyte JPEG image to a simple message type. To isolate the test scenario from other components in a system, the tested Web Service exchanged SOAP messages without introducing database and any business logic on the service-side.

The rest of this paper is organised as follows: Section 2 presents a brief discussion of the observed performance characteristics of the SOAP protocol and some related work on optimising the performance of the SOAP protocol. Section 3 explains the goals of this research work, including the decisions that were made and the anticipated benefits of this proposal. Section 4 provides an overview of TDXML and explains the techniques being used. Section 5 presents the evaluation results and comments on the performance characteristics of TDXML compared to the ASP.NET Document/Literal SOAP reference implementation. Conclusions and future work are discussed in Section 6.

2. Background and Related Work
Contrary to other people's findings [16, 20], this research's prior studies [28, 29] found that SOAP is not a particularly slow protocol, at least for good implementations such as the Microsoft ASP.NET document/Literal implementation. Tests run on a 1Gbps high bandwidth network show that SOAP offered latency times that were competitive to a fast binary protocol in most cases, only getting noticeably slower with large messages containing many XML tags. However, when the size and complexity of the messages increases, the performance of SOAP implementations gets comparatively worse, especially for the RPC/Encoded implementations. More important, the studies confirm that the performance of SOAP is affected by the following: (1) the use of Document/Literal encoding style works consistently faster than the RPC/Encoded encoding style; (2) the processes of serialisation and deserialisation are the primary bottlenecks when processing large SOAP messages; and (3) when the message structure gets more complex, the performance of SOAP gradually starts to get worse than large message sizes with simple structure.

Caching [9, 10, 18, 38] is one of the popular techniques used at the client, server or dedicated hardware to enhance the performance of SOAP. For example, the LYE technique proposed by Andersen et al. [9] used the caching technique on the server when marshalling the SOAP envelope and the SOAP response. Another popular technique is to reduce the network bandwidth requirement by using compact XML tags [35], compression [11, 21], Binary XML [14] or binary metadata [39]. A number of XML compression techniques have also been tried, namely gZip, XMll, and Millau [22]. All XML compression techniques produce high compression ratio. However, real time implementation performance figures of using the gZip compression technique working in conjunction with SOAP reveal that compression is only beneficial in situations where there is limited available network bandwidth [29]. Seshasayee et al. [36] proposed the SOAP-binQ middleware engine capable to serve SOAP messages with built-in Quality-of-Service that uses conversion handlers to replace SOAP’s XML representations with binary ones. There is concern to this proposal that the extra processing in XML/PBIO format conversion may affect the real time performance for devices with low CPU processing capabilities. Another major optimisation technique is to improve the parsing and serialisation/deserialisation processes by optimising the SOAP engine (e.g. XSOAP [12]) and the XML parser (e.g. XPP [37]). The trend in the last few years has been to forgo the

![Figure 1: An example showing a person instance represented using (a) XML and (b) TDXML](image)

A test scenario was developed to verify that TDXML is able to offer better performance than conventional SOAP XML encoding. A Web Service was built and tested using Microsoft’s ASP.NET 1.1 SOAP implementation and used to evaluate the performance of TDXML. In the tests, a TDXML message was embedded in the form of a string object in a SOAP message. The Microsoft .NET Remoting technology was also used to verify the interoperability of TDXML with other transport mechanisms. The performance of TDXML compared to the ASP.NET Document/Literal SOAP reference implementation. Conclusions and future work are discussed in Section 6.
DOM in favour of streaming interfaces such as the Simple API for XML (SAX) for faster parsing. However, technologies such as XML Query and XPath require the support of backward traversal, which makes using these technologies in streaming contexts difficult.

MTOM is a derivative of the ‘SOAP messages with attachments’ proposal (SwA) [30] and selectively encodes portions of a SOAP message using XML-binary Optimised Packaging (XOP) [26] to efficiently serialise XML Infosets containing binary data. Major vendors like Microsoft and IBM have started to offer support for MTOM in their products which make it a popular choice in the area of SOAP optimisation. The MTOM scheme appears to be an effective technique when used with large binary attachments, but the overhead of adding XOP MIME elements to the SOAP message means that it is less effective in messages with a large number of smaller binary components, such as those that are introduced when using WS-Security signatures [29].

3. Research Goals

It was observed that the majority of performance enhancement techniques require specially written protocol handlers or agents residing on both the sender and recipient ends in order to boost the performance. The use of specially written agents is inevitable if we want to achieve seamless integration with the SOAP protocol. However, problems start to arise when a particular agent introduces the issue of interoperability and increases the cost of implementation. Even when a certain level of performance gain can be achieved, the costs of implementation may easily out-weight the gain due to interoperability issue. Some of the proposals have given up the goal of meeting the XML Schema and XML InfoSet compatibility requirement [31], or otherwise require domain specific knowledge to implement the solution [2]. These proposals will make meeting the interoperability requirement of Web Services difficult. Furthermore, modifying base XML technology by removing certain XML features or adding binary semantics into XML [1] is likely to cause integration problem. Some of the proposals available today [31, 33] have not considered the WS-Security requirements and this will most likely pose difficulties for the proposals and stop them being adopted widely. Some proposals [35] address the message footprint aspect of Web Services only but have totally or partially overlooked the latency requirement. Furthermore, some proposals [6-8] require some specific set-up tasks to be performed before the actual consumption of a Web Service could be started.

TDXML is developed with a set of goals that avoid the unfavourable situations mentioned in the above and deliver the following desirable properties for an efficient XML document process:

(1) Based on the XML technologies. This means TDXML will be compatible with all the existing and future XML related technologies.

(2) Compact message footprint. The resultant message generated by TDXML must be compact. TDXML will use a specially designed indexing system to uniquely identify the occurrence of each XML element and attribute instances and thus avoid coding lengthy and repetitive nametags in an XML document.

(3) Improved parsing efficiency. TDXML should be able to improve the parsing efficiency in the following ways: (a) remove any nesting of the XML element structure by organising data items into a one-dimension flat table structure; and (b) allows direct access of data items and one pass parsing by using unique identifier for each individual element.

(4) Improved serialisation and deserialisation performance. The serialisation and de-serialisation processes in TDXML should involve the coding and decoding of identifiers for each data element rather than the unproductive effort of placing XML element tags and values in the correct position of an XML document. Also, the design of TDXML should enable the serialisation and deserialisation processes to be performed by going through the document once only.

(5) Support the encapsulation of binary data representation. TDXML should adhere to the MTOM design to ensure that any MTOM compatible encoding mechanism should also interoperate with TDXML.

(6) Support incremental update. The design of TDXML should allow new constructs to be defined easily so as to accommodate future XML technologies evolvement and new XML data types. This will ensure the interoperability capability of TDXML with existing and future XML technologies.

4. TDXML Overview

TDXML supports text-based XML encodings and adheres to all properties set by the XML specification. The extensibility feature of XML allows any portion of an XML document to be formatted using TDXML encoding scheme. A TDXML document is marked by the &lt;TDXML:Envelope&gt;&lt;/TDXML:Envelope&gt; block (&lt;TDXML:Env&gt; &lt;/TDXML:Env&gt; in short form). TDXML documents can be constructed in a number of ways. This paper introduces the Aggregated Scheme (also known as Schema-Content scheme) which uses the following two sub-sections to describe the data:

(1) Data Schema provides the structure of the XML document transmitted; and
(2) Content Table contains the index and the value of each XML element and attribute.

$<\text{TDXML}:\text{Env}>
<\text{TDXML}:\text{DS}>
\text{employees}[0]::=\text{SET OF person}
\text{person}[0]::=\text{SEQUENCE OF [}
\text{gender}(0)\ [\text{ATTRIBUTE}]\ \text{UTF8String} \ (\text{"M"})
\text{firstname}[0] \ \text{UTF8String},
\text{lastname}[1] \ \text{UTF8String},
age[2] \ \text{Integer}
</\text{TDXML}:\text{CT}>
</\text{TDXML}:\text{DS}>
</\text{TDXML}:\text{Env}>

\text{Figure 2: An example showing a block of TDXML data representing the details of two people.}

An example of a TDXML document encoded using the Aggregated Scheme is given in Figure 2. A TDXML Data Schema is marked by the <TDXML:DataSchema> block (<TDXML:DS> in short form). TDXML makes use of Abstract Syntax Notation One (ASN.1) [19] to describe the data schema contained in a TDXML Data Schema block. Square brackets ‘[ ] ’ are used to denote indexes for the elements and parentheses ‘( )’ are used to denote the indexes for attributes. The example showed in Figure 2 defines the simple structure of a set of <employees> containing two <person>s. Each <person> contains an attribute <gender> that has a default value ‘M’. Each <person> contains other details such as <firstname>, <lastname> and <age>. The root element <employees> is assigned tag [0]. TDXML uses the append-to-right style of numbering format to address each element and attribute. The <person> element is assigned [0][0] which is the first child element contained within the <employees> element. The attribute <gender> is assigned tag [0][0][0], which is the first attribute for the element <person>. The child elements of <person>: <firstname>, <lastname> and <age>; are assigned tags [0][0][0], [0][0][1] and [0][0][2] respectively. The unique identification of the first occurrence of <firstname> is [0][0][0][0], meaning the <firstname> element of the first occurrence of the element <person>, of the root element <employees>. The unique identification [0][0][1][1] points to the <lastname> element of the second occurrence of element <person> of the root element. The unique identification [0][0][0][0] points to the attribute <gender> of the first <person> element.

TDXML Content Table is identified by the <TDXML:ContentTable> block (<TDXML:CT> in short form). The example showed in Figure 2 contains the entry ‘[0][0][0][0]M’ which identifies the attribute <gender> in the first occurrence of the <person> element. The value of the attribute is ‘M’. The second row’s entry is ‘[0][0][0][0]Alan’ which identifies the child element <firstname> of the first occurrence of the <person> element. The value of the element is ‘Alan’.

4.1 Optimisation Techniques Utilised in TDXML

The time taken to serialise an object can be broken down roughly into the following components:

- \text{tsObjectMap} is the time taken to walk through the object.
- \text{tSerialise} is the time taken to produce a serialised string for a particular component of the object.
- \text{tXMLTag} is the time taken to write the XML start and end tags for a particular element and associated attributes.
- \text{tXMLDoc} is the time taken to write the XML output into the correct location within an XML document. This is dependent on the size and complexity of the resultant document.

Figure 3: An example showing two Content Tables are equivalent to each other so long as the tag-and-value pairs are identical.

\text{tsObjectMap} and \text{tSerialise} depend very much on the processing speed of the hardware and software platform as well as the size of the original object involved. TDXML did not make any optimisation in these two components. TDXML’s focus is to reduce the \text{tXMLTag} and \text{tXMLDoc} components by using the following techniques: (1) using only one short tag to identify each element and attribute; (2) the tags and their corresponding values are arranged into a single dimension tabular format instead of nested nodes; (3) the physical sequence of the tag-and-value pairs is irrelevant. That means the two Content Tables showed in Figure 3 are equivalent to one another.
On the receiving end, the time taken to deserialise the value string back into an object’s machine internal representation can also be broken down roughly into the following components:

- **tdObjectMap** is the time taken to map the object’s internal representation to the XML elements and attributes. This is very much depends on the type of parser involved. DOM parsers require walking through the object while SAX parsers do not.

- **tdXMLdoc** is the time taken to read the XML document for either validation purpose or generate some kind of data structure representation of the XML to be used by the application. A general purpose XML parser usually requires going through the same document more than once.

- **tdXMLtag** is the time taken to extract the serialised value associated with an XML element or attribute.

- **tDeserialise** is the time taken to perform actual conversion into machine internal representation. Again, tDeserialise is related mainly to the hardware and software platform. Hence, TDXML did not make any contribution in this component. TDXML reduces the tdObjectMap component by using schema specific parsers and precompiled serialiser/deserialiser. This technique is in line with other research outcomes [5, 13] which greatly reduce the time required to traverse through an object. TDXML is based on the observation that, if exploited correctly, schema information contained in a document should speed-up the lexical analysis and parsing of XML document. Therefore, TDXML reduces the tdXMLtag component by restricting the occurrence of a tag that can appear only once in a Content Table. Each tag-and-value pair is unique in a Content Table. This simplifies the effort required to validate a Content Table. More importantly, the tags become unique identifiers linking each serialised value to a designated deserialisation handler. Furthermore, to reduce the tdXMLdoc component, TDXML uses numbered short tags to reduce the overall document size as well as requiring the parsing routine to go through the input document only once. Hence, the time required to read the document is greatly reduced.

4.2 Integration Issues

The processing architecture of TDXML shown in Figure 4 demonstrates that TDXML allows different TDXML handlers to interact directly with the transport handler layer with or without a SOAP handler or an XML parser. The pipes are dedicated callback routines to handle TDXML specific tasks. At present, these TDXML handlers are hand coded to provide schema and transport specific functionalities. In the future, tools will be developed to generate the handlers automatically to become flexible plug-ins for use in other SOAP implementations. With the advent of XML-binary Optimised Packaging (XOP) standard [26], binary data can now be encoded more efficiently within an XML document by defining a convention of serialising XML Infosets that have certain types of content. TDXML can encapsulate binary image data using the MIME attachment standard with or without the use of XOP. It is totally up to the designer to decide whether to use TDXML with or without a SOAP implementation. The following section will look into this issue.

![Figure 4: TDXML processing architecture diagram showing TDXML handlers can interact directly with the transport layer.](image-url)

5. Performance Evaluation

TDXML is versatile yet simple to implement over any transfer protocols and provides an efficient alternative message packing mechanism. In this research, TDXML was implemented using Microsoft’s ASP.NET Document/Literal SOAP encoding style and .NET Remoting. The Microsoft .NET platform was chosen because it is one of the most efficient commercially available SOAP XML implementations. Using an efficient SOAP implementation as the reference implementation for TDXML gives a good indication whether TDXML is an efficient and compact optimisation mechanism for Web Services.

The goals of this evaluation were: (1) to investigate whether it was feasible to implement TDXML using an existing commercial SOAP implementation and (2) to investigate the performance characteristics offered by TDXML.

A set of customer and product information messages were defined that represented typical messages used by retail applications. The message
structures and sizes ranged from short to complex. A set of schema specific TDXML serialisers and deserialisers were developed in the C# language to handle each message type separately. C# handles event callbacks with its delegate objects to define reference types that can be used to encapsulate different schema specific serialisation/deserialisation methods.

5.1 Experimental Platform and Design

A simple client-server application with effectively no application logic on either side was designed to avoid any unnecessary server-side processing overhead that may affect the measured performance, such as database accesses and business calculations. The test set-up consisted of the following:

1. A multi-threaded test driver written in Microsoft’s C# language using the ASP.NET and .NET Remoting APIs. This client just constructs a request (SOAP or Remoting) with the identification field set to “REQUEST”, sends it to the server and waits for the response.
2. A target service, also written using C#, ASP.NET (hosted by IIS) and .NET Remoting (user written Remoting server), that receives the test message, deserialises the content and returns the original message with its identification field changed to “RESPONSE”.

A configuration file was used to control the set-up of the test environment, specifying attributes such as the number of client threads to be used, the duration of the test run and the message and encoding methods to be used. TDXML was implemented in the tests using a string object embedded within a SOAP request or a .NET Remoting request.

5.2 Test Scenario and Message

Five test messages were used in the evaluation study, in order to see how TDXML performed with messages of varying length and complexity. The first test message represents short text messages containing a text message of 50-byte length. The second message (simple) contains a single customer’s account record and uses string, Boolean and datetime data items. This is to represent typical request for single customer update or inquiry. The third message (medium) consists of twenty customer account records, representing a batch inquiry and subsequent update transaction. The fourth message (complex) consists of one customer record and 50 product details, representing an invoice or a customer statement. The fifth message (simple/jpeg) added a 14 Kbyte binary JPEG image, representing requests with a photograph of a customer or product to a simple message. This will give an idea how TDXML handles messages which are predominantly containing binary opaque data.

Four groups of tests were conducted to evaluate the impact of TDXML on existing SOAP and binary implementations. The first set of tests used TDXML handlers to serialise/deserialise different test messages to/from string objects which were encapsulated in a Web Service using ASP.NET Document/Literal SOAP encoding style for communication between a set of client-server machines with concurrent consumer load and network bandwidth of 1Gbps. The second set of tests also used TDXML handlers to serialise/deserialise test messages to/from string format but used the .NET Remoting HTTP/binary formatter mechanism to communicate between the client-server machines. The third set of tests used ASP.NET Document/Literal SOAP encoding to serialise/deserialise the test messages directly and the fourth set of tests used the .NET Remoting HTTP/binary formatter to serialise/deserialise the test messages.

5.3 Performance Metrics and Measurement

The following performance measurements were taken during the tests and used to evaluate the performance, resource usage and scalability characteristics of TDXML.

1. Latency. The tests measured the round-trip time taken to send a single message and receive a response, from the test driver to the server and back to the waiting test driver.

2. Processor and Bandwidth Utilisation. The tests also measured processor utilisation on both client and server machines and observed the bandwidth required to send and receive each test message.

3. Throughput. A set of tests was conducted by varying the number of concurrent client loads from 4 to 32 threads to find the peak throughput available from each implementation. These tests use a number of concurrent driver threads to send large volumes of concurrent messages to the SOAP servers being tested and record the number of round-trips completed each second.

5.4 Performance Analysis

Two Dell 6650 servers were used as both client and server systems. The hardware and software configuration of these systems was:

- Four 1.6GHz Intel Xeon MP processors
- 3669Mbytes of memory
- Microsoft .NET Framework 1.1
5.4.1 Message Size Analysis

Table 1: Observed message sizes on the wire for different message types

<table>
<thead>
<tr>
<th></th>
<th>SOAP</th>
<th>TDXML</th>
<th>Remoting</th>
<th>TDXML Remoting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short</td>
<td>375</td>
<td>422</td>
<td>318</td>
<td>228</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-12.5%</td>
<td>15%</td>
<td>39%</td>
</tr>
<tr>
<td>Simple</td>
<td>867</td>
<td>663</td>
<td>726</td>
<td>472</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24%</td>
<td>16%</td>
<td>46%</td>
</tr>
<tr>
<td>Medium</td>
<td>7588</td>
<td>4712</td>
<td>3479</td>
<td>4524</td>
</tr>
<tr>
<td></td>
<td></td>
<td>38%</td>
<td>54%</td>
<td>40%</td>
</tr>
<tr>
<td>Complex</td>
<td>21149</td>
<td>12101</td>
<td>6156</td>
<td>11910</td>
</tr>
<tr>
<td></td>
<td></td>
<td>43%</td>
<td>71%</td>
<td>44%</td>
</tr>
<tr>
<td>Simple</td>
<td>21195</td>
<td>20483</td>
<td>15650</td>
<td>20291</td>
</tr>
<tr>
<td>Jpeg</td>
<td></td>
<td>3%</td>
<td>26%</td>
<td>4%</td>
</tr>
</tbody>
</table>

Table 1 shows the size of the messages seen on the wire and the amount of reduction in size given by the alternative TDXML implementations under test. There are three factors (size, complexity, and binary data) affecting the effectiveness of TDXML and the relative effects of these can be seen in the results.

For short message type with 50 byte length text string, TDXML implemented in SOAP increases the resultant message size by 12.5% because, on top of the original SOAP XML namespaces, there are extra tags such as <TDXML:CT> and TDXML index '[0][0]' added. However, when TDXML is implemented in .NET Remoting, the SOAP namespace overheads are removed. Also, it was found that Remoting’s binary representation for a short message takes more storage than the corresponding TDXML representation. This makes TDXML the best performer in short message type. For simple message type, TDXML is also the best performer. TDXML yields 24% reduction in size in the case of SOAP implementation and 46% reduction in .NET Remoting. This shows that when the structure of a message is simple and the size is small, TDXML is the best choice in terms of storage requirement.

For medium message type, TDXML is able to offer 38% reduction in size in the SOAP implementation and 40% reduction in the .NET Remoting implementation. However, .NET Remoting binary representation becomes more efficient (54% reduction) than TDXML. The same trend appears for the complex message type, where TDXML SOAP implementation offers 43% reduction and TDXML .NET Remoting implementation offers 44% reduction, but .NET Remoting binary offers 71% reduction. This shows two things: (1) TDXML is able to reduce the size for medium and complex SOAP messages significantly (over 40%); and (2) When a message is composed of large number of repetitive structure (i.e. 20 customers or 50 product lines), TDXML is less efficient in network bandwidth requirement than binary representation. For the simpleJpeg message type where a 14Kbyte JPEG image is embedded in a simple message type, TDXML offers very little reduction in size (only 3%) because the resultant message is dominated by the Base64 encoded JPEG portion. However, since TDXML supports the MIME multipart mail encoding scheme; TDXML (15773bytes) is able to offer the same level of reduction as the .NET Remoting binary representation (26%) when TDXML is implemented with MIME.

In summary, these tests show that TDXML offers significant reduction in messages size when compared to text-based XML, in particular when the message content is complex containing large number of different fields. Also, when the message is dominated by large amount of binary opaque data, TDXML’s MIME implementation offers significantly smaller message sizes when compared to Base64 encoded SOAP implementation.

5.4.2 Latency Analysis

Figure 5 shows the average latency recorded for each message type. The TDXML MIME Remoting test case only applied to the simpleJpeg message type since all other test messages do not contain large binary opaque data. These tests were run over a lightly loaded 1Gbps LAN. This means the latency figures obtained were mainly influenced by the processor time needed on the client and server systems, rather than by the time that the messages take to traverse the network. Changing to a lower bandwidth edge network should further increase the competitiveness of TDXML for its bandwidth-conserving capability.

In the ASP.NET scenario, even with the overheads incurred by the SOAP handler in serialising and deserialising the TDXML string object, TDXML is able to offer the same level of latency performance as ASP.NET Web Services for short and simple message types. When the message size and complexity increase, the benefits offered by TDXML start to payoff. TDXML offers 22% improvement in latency for the medium message type and 29% improvement for the complex message type when using the ASP.NET Document/Literal SOAP implementation.
The latency picture for TDXML using the .NET Remoting with HTTP/Binary encoding scenario is somewhat different. It seems there is a relationship between message size and latency. It was observed that TDXML offers smaller resultant message size than .NET Remoting for short and simple message types and similar size for the medium message type. Even though TDXML requires .NET Remoting to serialise and deserialise the string object first before it could perform serialisation and deserialisation, the latency figures offered by TDXML are better than or of similar level to .NET Remoting for short, simple and medium message types.

However, for the complex and simpleJpeg message types, the message sizes for TDXML are larger than their corresponding .NET Remoting figures. Therefore, the latency results offered by TDXML for complex and simpleJpeg message types are slower than .NET Remoting. (28% for complex message and 68% for simpleJpeg message type). However, when TDXML was implemented using MIME for the simpleJpeg message type, the result was quite impressive (6.8 ms versus Remoting’s 5.6 ms). When comparing the latencies offered by TDXML implemented with .NET Remoting against ASP.NET SOAP, we can observe improvements of 40% for complex message type, 44% for medium message type, 57% for simple message type and 88% for short message type.

In summary, when TDXML is implemented with a binary transport, it offers latency performance levels that are comparable to the binary transport for most message types (short, simple and medium messages) and is not far behind for other complex message types. As for message with large binary data, TDXML can offer similar level of performance as the binary transport with the use of MIME encoding scheme.

5.4.3 Throughput Analysis
To obtain the maximum throughput that each implementation can handle between the client and server, the tests described in the previous section were repeated using a varying number of concurrent threads (4, 16, and 32 threads) to measure the maximum number of requests that the server can handle in a set time period. Maximum throughputs were achieved at 4 and 16 client threads. The results of these tests are reported in messages per second (msg/sec).

From Table 2, we can see that TDXML consistently offers equal or higher throughput figures than ASP.NET SOAP implementation for all message types. Although there is not much gain in the short, simple and simpleJpeg message types, however, TDXML is able to offer 23% improvement in the medium message type, and 63% in the complex message type. When TDXML was implemented using .NET Remoting, the improvements over ASP.NET SOAP are even more: over 60% in the short and simple message types, over 40% in the medium message type, and 64% in the complex message type. TDXML using .NET Remoting and MIME implementation is able to offer 45% improvement than ASP.NET SOAP in the simpleJpeg message type. These figures show that
TDXML is able to offer better throughput than ASP.NET SOAP implementation in all message types.

Table 2: Throughputs offered by SOAP and TDXML

<table>
<thead>
<tr>
<th>msg/sec</th>
<th>SOAP</th>
<th>TDXML Soap</th>
<th>TDXML Remoting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short</td>
<td>543</td>
<td>550</td>
<td>868</td>
</tr>
<tr>
<td>Simple</td>
<td>471</td>
<td>507</td>
<td>814</td>
</tr>
<tr>
<td>Medium</td>
<td>212</td>
<td>261</td>
<td>298</td>
</tr>
<tr>
<td>Complex</td>
<td>78</td>
<td>129</td>
<td>128</td>
</tr>
<tr>
<td>Simple Jpeg</td>
<td>170</td>
<td>170</td>
<td>197</td>
</tr>
<tr>
<td>Simple Jpeg Mine</td>
<td>N/A</td>
<td>N/A</td>
<td>247</td>
</tr>
</tbody>
</table>

5.4.4 CPU Utilisation Analysis

To investigate whether TDXML’s serialisation and deserialisation overheads are CPU intensive or not, a heart-beat routine was developed to monitor the server’s CPU utilisation (percentage) at a fixed interval for each test run. Average figures were calculated at the end of each run.

In general, for the single client latency tests, the measured CPU utilisation figures for TDXML were at all times kept below 19% for all message types with the highest demand noted for the simple/jpeg message type (average 11%). This is because the Base64 conversion process of the 14Kbyte JPEG data requires more CPU and memory resources.

6. Conclusion and Future Work

This paper has systematically studied a variety of real-time performance characteristics (latency, throughput, and CPU utilisation) and network bandwidth requirement (message size) of TDXML, a newly proposed efficient and compact message encoding mechanism for XML-based Web Services. The tests are based on a set of scenarios designed to reflect typical commercial situations. All the test results demonstrate that TDXML is able to enhance the performance of SOAP by 24% to 46% reduction in message size and 22% to 88% improvement in latency. Especially, the evaluation tests show that TDXML implemented using ASP.NET Document/Literal encoding can significantly reduce the network bandwidth and round-trip latency figures for medium and complex message types. When comparing TDXML to an efficient binary encoding mechanism such as .NET Remoting HTTP/Binary encoding, TDXML offers equal and even slightly better performance than .NET Remoting for short, simple to medium size message types due to more compact message footprints. However, TDXML still lags behind .NET Remoting by a margin of 28% when the message size becomes larger and message structure becomes more complex. This is an area that we will focus on for further improvement.

There is more work required for TDXML before the wider Web Services community can adopt it. Further performance validation works are required for other implementation environment such as Java Web Services. The interoperability capability of TDXML with other WS-* protocols such as WS-Security needs to be verified. Also, development tools such as TDXML pre-compiler and APIs need to be made available for developers to use.

References


[18] Devaram, K. and Andresen, D., SOAP Optimization via parameterized client-side caching, Department of Computing and Information Sciences, Kansas State University, 2003


[31] Nielsen, H.F., Sanders, H., Christensen, E., et al., Direct Internet Message Encapsulation (DIME), Internet Draft, Microsoft, 17 June 2002


