Optimizing Web Services Performance by Differential Deserialization

Toyotaro Suzumura, Toshiro Takase and Michiaki Tatsubori
Tokyo Research Laboratory, IBM Research
1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502, Japan
{toyo, e30809, tazbori}@jp.ibm.com

Abstract

Web services technology has emerged as a key infrastructure that enables business entities to interact with each other without any human inventions. In order for the technology to be widely used, especially in any field where a large volume of transactions may be processed, it is highly desirable that the Web services engine should tolerate such environments. In this paper, we present a novel approach for improving Web services performance. We first focus on the fundamental characteristics of the Web services in that the SOAP messages on the wire are mostly generated by machines and have a lot of similarities among the processed messages. By making use of these features and eliminating the redundant processing, we propose a new deserialization mechanism that reuses matching regions from the previously deserialized application objects from earlier messages, and only performs deserialization for a new region that would not be processed before. Through our experiments in this paper, we observed that our approach obtained a 288% performance gain (maximum) by incorporating the differential deserialization into the Axis SOAP engine.

Key Words: Web Services, Performance, SOAP, JAX-RPC, Deserialization, SOA, XML

1. Introduction

Currently, Web services are spreading widely throughout the world. Web services are an enabling technology for interoperability within a distributed loosely coupled and heterogeneous computing environment. The Web services technologies are built on SOAP as the messaging layer, WSDL as the interface description, and UDDI as the service discovery mechanism. Without any doubt, it is clear that the Web services will be a key building block for using the next generation computing platforms such as SOA (Service Oriented Architecture) and Grid computing.

Over the last few years, several studies have investigated the Web services in order to make viable Web services technologies to replace traditional distributed object technologies such as CORBA. Those studies have mainly focused on the poor performance and proposed several approaches for improving the overall performance without compromising the interoperability. The poor performance stems from the fact that the Web services are based on the XML-based communication protocol, SOAP. SOAP provides the fundamental messaging infrastructure supporting XML document exchange and Remote Procedure Calls using XML messages, but its redundant characteristics and its textual representation is a major performance bottleneck.

It is not straightforward to optimize such XML processing in a general manner, but we could optimize the processing by making use of the characteristics of the specific problem domain. In a Web services architecture, most of the messages on the wire are generated by machines. In particular, RPC-style request-response messages are often generated by middleware with XML serializers. When accessing Web services in client code, proxy classes and frameworks provided by middleware handle all of the infrastructural coding. Though formatting styles are different for various programming languages, implementation vendors, or versions, the same XML serializer implementation generates the same kind of service requests and responses with different parameters and return-values in similar byte sequences. This is because this kind of XML serialization is performed by a certain runtime library or proxy code and generated by a particular tool provided by middleware or a development environment.

In [1], we proposed a new approach to improve the performance of an XML parser based on the fundamental characteristics of Web services. The proposed XML parser efficiently detects the

IEEE COMPUTER SOCIETY

differential regions between a new XML message and the past messages, and then partially parses only the differential portions. Given a new XML document as a byte sequence, the parser remembers the byte sequences using a DFA (Deterministic Finite Automaton), where each state transition has a part of a byte sequence and its resultant parse event. In addition, it remembers processing contexts in the DFA states so that it can partially parse the unmatched byte sequence until it meets a resultant state from which it can transit to existing states.

In this paper, we explore the performance impact by extending the approach presented in [1] to the deserialization framework in the Web services architecture. Deserialization is a process of converting XML messages to application objects passed to application logic. The deserialization involves a series of tasks such as fetching an appropriate deserializer from the type mapping registry, and constructing a Java object from an XML message. The cost of object creation is higher as the object becomes more complex and the object tree becomes deeper. We can then eliminate those tasks by only processing the new regions of the XML messages and reuse the constructed objects deserialized in the past. In this paper we will also investigate the best strategy for reusing the application objects, since the application objects are not necessarily read-only and might be modified by other components.

The rest of the paper is organized as follows. In Section 2, we introduce some related work for optimizing the Web services performance and possible optimizations for the deserialization component in the architecture. In Section 3, we propose an overview and the architectural design of the differential deserializer. Section 4 describes the design considerations for object reuse, and Section 5 describes the performance study. Finally we conclude this paper with Section 5.

2. Optimizing Web Services Performance

This section describes the related work and investigates possible approaches for optimizing the deserialization process in the Web services architecture.

2.1 Related Work

Several studies have proposed a variety of approaches for optimizing Web services performance.

Sender-side Optimization:

[3] proposed an approach for removing the serialization bottleneck. Their approach is to avoid complete serialization of SOAP messages by storing and reusing message templates. The idea is to perform a complete serialization only when the first message of a certain structure is sent by a SOAP communication endpoint. Subsequent messages with the same structure and some of the same content can then reuse parts or all of the saved template instead of regenerating it from scratch. The paper [2] described a response cache mechanism for Web services client middleware using several approaches such as caching the post-parsing representation and caching application objects.

Optimization for Scientific Computing:

The scientific grid project is an area of high performance computing that is adopting the Web services architecture. Since XML primarily uses ASCII as the representation format for data, sending scientific data (for example large arrays of floating point numbers and complex data types) via standard implementations of SOAP can result in a severe performance penalty. [13] proposes an approach that sends arrays as Base64 encoding supported by SOAP as a semi-binary encoding of the parameters. [8] proposed an optimized version of SOAP called XSOAP. The authors built a new XML parser that is specialized for SOAP arrays, improving the deserialization routines. This study employs HTTP 1.1, which supports chunking and persistent connections.

Binary Web Services:

Fast Web services [21] and other binary XML protocols [20][22][23] have been proposed for enhancing Web services performance. However this is applicable in restricted computing environments where each sender or receiver knows the format of the binary protocol, and does not consider interoperability unless there is some mechanism to publish the information about how the container support the binary protocol exchange. However systems like UDDI currently do not have such a mechanism.

2.2 Optimizing Deserialization

To summarize the optimization technologies described in the previous section, they ignored deserialization even though some research had pointed out that serialization and deserialization greatly affect overall performance. Some studies [6] [8] [10] performed interesting performance evaluations on the serialization and deserialization. Based on experiments in [6], they observed that the deserialization overheads for processing incoming messages are higher than the serialization overheads. The Document/Literal implementation takes 23.8% of the total time for deserialization and only 10.7% for serialization. Next we describe the deserialization mechanism defined in

JAX-RPC (Java APIs for XML-based Remote Procedure Call) [11].

Deserialization Mechanism in JAX-RPC

Deserialization is the process of reconstructing the object from the XML data. The serialization and deserialization mechanics in JAX-RPC rests on the availability of type mapping systems defined in a registry. A type-mapping system is the core component to make the serialization or deserialization work. When the SOAP engine reads a particular piece of XML and comes across a given element of a given schema type, it can locate an appropriate deserializer in order to convert the XML into Java. The SOAP engine usually has a registry where a set of required type mappings are registered. JAX-RPC introduces layers called *TypeMapping* and *TypeMappingRegistry* that contain multiple *TypeMappings*, and then the *TypeMapping*s allow you to map XML and Java types.

Deserializing an XML message into Java objects involves the following steps:

1. Open the XML element that represents the object
2. Recursively deserialize each of the object's members which are encoded as sub-elements after locating an appropriate deserializer from a type mapping system
3. Create a new instance of the Java type, initializing it with the deserialized members
4. Return the new object

Currently no research work is attempting optimization of the deserialization process, even though [3] investigated the serialization process (as already mentioned). Even though serialization and deserialization are symmetric functions, different issues need to be solved because the reused object is different. In the serialization process, the XML message is a target for recycling. In the deserialization process, the target is an application object. You cannot simply reuse the object because there could be some situations in which an application object is modified by applications. Therefore the issue is how an application object can be reused without any side effect while fulfilling the requirement that redundant object copying is avoided.

3. Differential Deserialization Architecture

This section describes our method for improving the SOAP implementation. We first introduce our previous work, a differential parser, and then describe our new approach called "*differential deserialization*".

3.1 Deltarser: Differential Parser

[1] proposes a novel mechanism for efficiently processing XML documents for most XML usages. Given a new XML document in a byte sequence, the proposed XML parser does not analyze most of the XML syntax in the document, but just compares the byte sequence with the ones that were previously processed. The parser then reuses the resultant parse events stored in previous processing. Only the differential parts from the previously processed documents are processed in a normal manner for XML parsing. The parser remembers the byte sequences using a DFA (Deterministic Finite Automaton), where each state transition has a part of a byte sequence and its resultant parse event. In addition, it remembers processing contexts in the DFA states so that it can partially parse the unmatched byte sequence until it meets a resultant state from which it can transit to existing states. Then it continues to transit in the DFA.

3.2 Differential Deserialization

In Web services architectures, the XML parsing cost as it affects overall performance is not a dominant factor. However the notion of skipping the processing for the previously processed part can be applied to other components in the Web services architecture. Since we found that the deserialization cost is relatively large, we apply the notion of differential processing to the deserialization process. The fundamental idea is to deserialize only the region that would have not been processed in the past, recycle the application object deserialized at the first processing, and reset the fields in the object. This approach, what we call "*differential deserialization*", can eliminate a series of processes required for the completion of deserialization, especially because we believe that eliminating the object creation should be effective.

Studies such as [14] point out that objects are expensive to create. Objects need to be created before they can be used and garbage-collected when they are finished with. The more objects you use, the heavier, this garbage-cycling impact becomes. Object recycling is a traditional performance-tuning method, especially for objects that are frequently used and discarded. Recycling can also apply to the internal elements of structures. Therefore we can apply such an object recycling approach for differential deserialization. .

3.3 Architecture Design

We illustrate an overall architecture that realizes the notion of our differential deserialization in Figure 1.

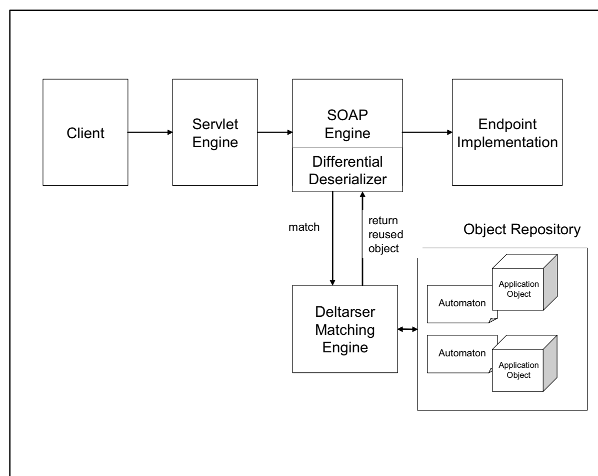


Figure 1. Differential Deserializer Architecture

The architecture is composed of the following components:

1. Servlet Engine,
2. SOAP engine,
3. Differential Deserializer,
4. Endpoint implementation,
5. Matching Engine, and
6. Object Repository.

The difference between this architecture and an ordinary Web services architecture is only that the deserializer is replaced with our differential deserializer. The differential deserializer is a component that communicates with what we call "Matching Engine". The Matching Engine is a component that provides all of the functionality required for differential deserialization. The component provides two functions: one is to dynamically generate an automaton from the incoming XML messages, and then after deserializing into the application object by using the SOAP engine in the normal way, it makes a link between the defined automaton and the application object. The second function is to match the incoming messages with the existing set of automaton paths and if matched, return the linked application object to the SOAP engine after partially deserializing only the region that differs from the past messages and then resets the fields. Next we give a more concrete description regarding the mechanism of the differential deserialization.

3.3.1 Creating New Deserialization Automaton

When the SOAP engine receives a brand new message, the Matching Engine creates a new automaton path (a "deserialization automaton") after the deserialization component in the SOAP engine generates the application object in the normal way. For instance, when dealing with the new SOAP message shown in Figure 2, the Matching Engine detects that there is no matched state transition by byte-sequence matching, and then starts to create a new deserialization automaton. Figure 3 shows a sample of the deserialization automaton and a series of state nodes with black representing the newly created deserialization automaton. After reaching the final state (*</SOAP-Envelope>*), the Matching Engine makes a link from the final state to the corresponding application object.

Note that a deserialization automaton consists of two states: a fixed state and a variable state. A fixed state is a state whose byte sequence is not changed such as a start tag (e.g. *<SOAP-Env:Body>*), an end tag (e.g. *</SOAP-Env:Body>*), or some text content that is defined as a constant value in the XML Schema. A variable state is a state whose byte sequence can vary in different messages. For example, the part between a start tag *<q>* and an end tag *</q>* is variable and should be represented as a variable state.

In our approach, a variable state is determined by checking a set of RPC (Remote Procedure Call) parameters defined in the SOAP envelope. The SOAP envelope object allows programmers to access the information with regards to what RPC parameters should be passed for certain SOAP operations and what their data types are. While creating a new deserialization automaton, the Matching engine collects information for variable states and creates a table called a Variable Table for maintaining them. This table is used for updating the fields with new values when reusing the application objects. Each record in the table contains the following information:

1. Variable ID: a key that identifies the variable object
2. Object parent: a target object that a new value of the variable object should be set to.
3. Class type: the data type of the variable object
4. Object value: the new value of the variable object
5. (Optional) Method setter: a setter method of the parent object that updates the new value.

The last one is optional because it is possible to obtain a setter method for updating the value by investigating the parent object with the Java reflection APIs, although it is more straightforward to preserve the setter method object. However, since it is not clear whether preserving the setter method object can help

IEEE COMPUTER SOCIETY

improve the performance, we will examine the performance impact in Section 5.

Finally after the Matching Engine creates the corresponding Variable Table, the engine also attaches the table with the final state of the deserialization automaton to the application object.

3.3.2 Object Recycling and Differential Deserialization

When the matching engine processes similar messages, the engine traverses the existing deserialization automaton. When going through a variable state, the byte sequence up to the next state is partially parsed and deserialized by the SOAP engine. Then the engine updates the new value in the Variable Table using the Variable ID. Finally if the engine reaches the existing final state after traversing the deserialization automaton, the matching engine knows that the engine should reuse the application object and resets its variable fields with the set of new values specified in the Variable Table. There are several approaches for reusing the application object. Which is the best approach depends on how complex the object is and how the object is accessed by the application, so this issue will be discussed in the next section.

4. Design Considerations for Object Recycling

In this section we discuss design considerations with regards to the issues raised in the previous section: the best approach for reusing application objects and the best approach for resetting the values to reused application objects.

4.1 Reusing Application Objects

There are two possible approaches for reusing application objects: reusing a reference to the application object or using an object that is replicated from the application object. It is important to consider the design for reusing application objects without causing any side effects for applications.

For the first approach, it is fairly straightforward as well as fastest, but is available only in a restricted case. The limitation is that a business object must be read-only except for primitive value such as String and Integer.

The second approach is safe and can be applied to various situations. The simple way to realize this approach is to clone the entire object tree, but sometimes a certain part is fixed and the structure would not change at all. In such a case, it would be significantly more effective to store only the specific portion of the application object that corresponds to the fixed part. Therefore, it is challenging to identify a method for dynamically adjusting the granularity of the reuse the business object. Since we are focusing on how the differential deserialization is effective for improving the performance of Web services containers, this granularity issue will be described in a later paper. There are four ways for reusing an application object [2].

```
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
 <SOAP-ENV:Body>
 <ns1:doGoogleSearch          xmlns:ns1="urn:GoogleSearch"
SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/enco
ding/">
  <q xsi:type="xsd:string">ICWS 2005</q>
  <start xsi:type="xsd:int">0</start>
  <maxResults xsi:type="xsd:int">10</maxResults>
  <filter xsi:type="xsd:boolean">true</filter>
  <restrict xsi:type="xsd:string" />
  <safeSearch xsi:type="xsd:boolean">false</safeSearch>
  <lr xsi:type="xsd:string" />
  <ie xsi:type="xsd:string">latin1</ie>
  <oe xsi:type="xsd:string">latin1</oe>
  </ns1:doGoogleSearch>
  </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

Figure 2. SOAP Messages for Google Web APIs

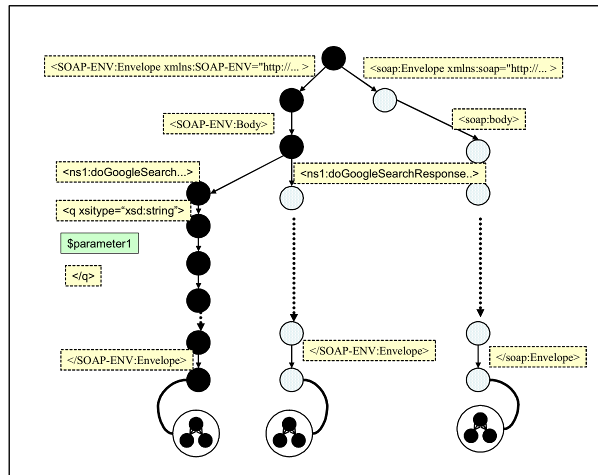


Figure 3. Deserialization Automaton

1. No copying

When we can guarantee that the object is read-only and the endpoint implementation will not change the

object, we can reuse a reference to the application object without any copying.

2. Copying by cloning

If the application object does not override the clone method of the Object class, calling a clone method of the object performs only a shallow copy operation. In this case, if all of the fields of the application object are immutable types, the clone method is adequate to avoid side effects. If some mutable fields exist in the application object, the shallow copy is not enough. If some mutable fields exist, a deep copy method which recursively copies the object tree is required. Currently the Java classes generated by the WSDL compiler do not implement the clone method, so we add a clone method that performs the deep copying.

3. Copying by the Java serialization method

Java serialization allows the target object and all of its fields to be serialized into a byte array, except for transient fields. Also, the objects referred to by its fields are serialized recursively. In Java deserialization, another new object of same type and having the same value is reconstructed from the serialized byte array. That is to say, we can copy the object deeply using Java serialization and deserialization. The Java serialization is only available when the application object implements the *java.io.Serializable* interface, but the classes generated by WSDL2Java do implement this interface.

4. Copying by using the Java reflection API

This approach performs a deep copy method by using the Java reflection API. This method can deeply copy bean-type and array-type objects. With this approach, application developers need not implement the clone method as in the second approach.

4.2 Setting new values

Another design issue is to consider how to reset the fields of an application object with new values as described in Section 3.3.2. There are two ways to reset the value against the target object as in the following.

i. Use the reflection API: By searching for the appropriate method for setting a new value for the target object with the Java reflection API,

ii. Preserve the parent object and a method: To avoid the searching cost in the above method, we can merely store the method object when creating a new automaton state node.

```
<ns1:doGoogleSearchResponse>
<documentFiltering> false </documentFiltering>
<directoryCategories
    xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
    xsi:type="ns2:Array"
    ns2:arrayType="ns1:DirectoryCategory[0]">
</directoryCategories>
<resultElements
    xmlns:ns3="http://schemas.xmlsoap.org/soap/encoding/"
    xsi:type="ns3:Array" ns3:arrayType="ns1:ResultElement[3]">
  <item xsi:type="ns1:ResultElement">
        <cachedSize xsi:type="xsd:string">12k</cachedSize>
        <hostName     xsi:type="xsd:string">     hostname
</hostName>
  </item>
  <item xsi:typ  ………. (abbreviation)
</resultElements>
  ……. (abbreviation)
</ns1:doGoogleSearchResponse>
```

Figure 4. A chunk of SOAP Message for the *GoogleSearchLarge* service

5. Performance Study

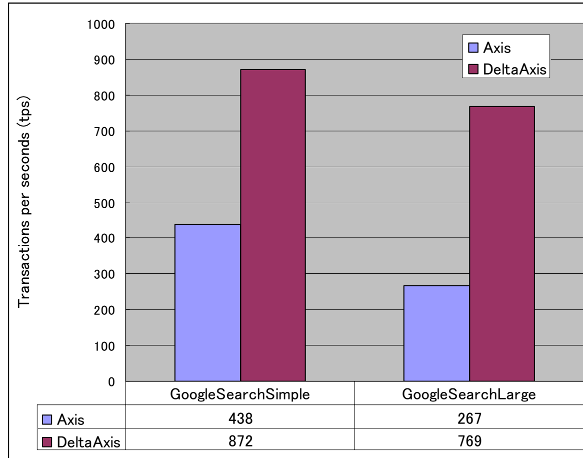
The experiment was conducted by incorporating the differential deserialization into Apache Axis [18] (version 1.2 beta 2), an open source implementation of the SOAP engine and we made the performance comparisons using the following implementations:

1. Axis without any change
2. Axis with the differential deserializer (we will call it *DeltaAxis* hereafter). In order to make performance comparison among the strategies for object recycling described in Section 4.1, we tested the implementations with the following strategies. Except for B, resetting the field to the application object is done by the preserved setter method. The reflection cost for the update (See 4.2) can be determined by comparing A and B.
   - A Copying by cloning
   - B Copying by cloning with the update of new fields by the Java reflection API
   - C No copying
   - D Copying by the Java serialization method
   - E Copying by the Java reflection API

Experimental Environment and Results

The experimental environments are using IBM JDK 1.4.2 as the Java virtual machine, Apache Tomcat 4.1.13 as the application server, and Xerces as the XML parser. For the server, we used Windows 2003 Server on an IBM xSeries 440 with a 2.4 GHz Xeon CPU and 4 GB of RAM. For the client, we used Linux (kernel 2.4.18) on an IBM IntelliStation M Pro with a 2.5 GHz Pentium 4 CPU and 2 GB of RAM, and the two machines are connected via a 1 Gb Ethernet, with an average latency time of 0.171 ms.

The Google Web APIs [17] was modified to perform the performance evaluation. To quantify how effective the differential deserialization is, we prepared two Web services. One is called *GoogleSearchSimple* to which a client sends a search request message of 1 KB and gets a dummy response message that contains



	GoogleSearchSimple	GoogleSearchLarge
Axis	438	267
DeltaAxis	872	769

Figure 5. Performance Comparisons between Axis and DeltaAxis

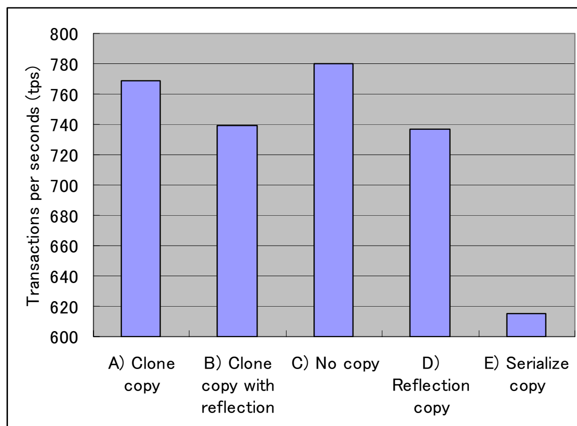


Figure 6. Performance Comparisons among various object recycling methods

nothing. The request SOAP message is the same as in Figure 2 and the corresponding application object does not have any nested objects inside but has only a set of primitive fields. For this Web service, the inside fields are all immutable so there is no need to care about the issues discussed in the previous section.

The other is called *GoogleSearchLarge* to which a client sends a large and complex message of 2 KB that is something like the response message returned from the Google search request. Part of the SOAP request message is illustrated in Figure 4. The top element has multiple nested elements and the corresponding application object has array objects and complex objects as the fields. With this Web service, various object recycling strategy can be examined.

Those two Web services do not prepare either the actual business logic or actual response messages, so we can measure only the time differences showing how much our approach optimizes the deserialization function.

Our experiment measured the throughput by running 10 threads that simultaneously send requests to the deployed Web services, with a total of 30,000 requests in each experimental run. This measurement is conducted after sufficient warm-up to exclude the compilation time of the JIT compiler. The throughput is defined as the number of transactions per second (tps). The results are presented in Figure 5 and Figure 6.

First, Figure 5 shows the total throughput for the two implementations, Axis and DeltaAxis to process two Web services. DeltaAxis is the modified Axis implementation with differential deserialization and object recycling by using the cloning method. For the *GoogleSearchSimple* service, DeltaAxis obtained almost double the throughput of Axis. For the *GoogleSearchLarge* service, DeltaAxis recorded 288% more throughput than Axis.

Next Figure 6 illustrates the comparisons among the different strategies for object recycling described earlier, while using the *GoogleSearchLarge* service. All of the strategies had more than 600 tps, which is over twice Axis (287 tps). First we see that the preserved setter method contributes to performance somewhat with the comparison between A and B. Second, the graph shows the significant performance disadvantage of the serialize copy E. The clone copy A obtains a great throughput number and is very close to the no-copy method C. The reflection copy is a bit slower than the clone copy but not a disappointing number. In summary, the performance ordering is C (no copy) > A (clone copy) > B (clone copy with reflection) > D (reflection copy) > E (serialize copy). However from the viewpoint of application developers, they need to add the clone-copy method for A, but the reflection copy D could be supported internally by the SOAP engines, which would enable the developers to make the most of the differential deserialization in a more transparent manner.

6. Concluding Remarks

In this paper, we have presented a new approach for improving the Web services performance by reusing an application object and performing partial deserialization only for the regions that differ from the messages processed in the past. This strategy is based on the fundamental characteristics of Web services in that SOAP messages are similar to each other.

Our experiments show the performance boosts that we achieved using these strategies. However, several important issues still remain open for further research.

First we will investigate cases where reusing the entire object tree is not efficient. We targeted the case where each message has exactly the same structure, but the larger messages on the wire tend to have some repetition of certain elements, such as for the *GoogleSearchLarge* service. In such a case, recycling the entire object tree is not an optimal solution since the repetition number might differ for each request. An optimal strategy would be to copy only the frame of the entire object and one object that corresponds to one iteration. There would be other cases where redundant object recycling could be eliminated. Thus we will investigate ways of dynamically adjusting the granularity of copying.

Second, we will work towards making this mechanism scalable in a cluster-like computing environment, where a front-end server dispatches a large number of client requests to multiple application servers. Reusing objects in such an environment would not be straightforward and requires us to consider issues such as thread safeness and scalability. Moreover we will need a mechanism like garbage collection for optimizing the size of automata as it grows.

Acknowledgements

We would like to thank Yuichi Nakamura and many other members of the IBM Tokyo Research Laboratory for many valuable discussions and helpful comments.

References

[1] Toshiro Takase, Hisashi Miyashita, Toyotaro Suzumura, and Michiaki Tatsubori. An Adaptive, Fast, and Safe XML Parser Based on Byte Sequences Memorization, to appear in the *14th International World Wide Web Conference* (WWW 2005)

[2] Toshiro Takase and Michiaki Tatsubori, Efficient Web Services Response Caching by Selecting Optimal Data Representation, *The 24th International Conference on Distributed Computing Systems* (ICDCS 2004)

[3] Nayef Abu-Ghazaleh, Michael J. Lewis, Differential Serialization for Optimized SOAP Performance, *The 13th IEEE International Symposium on High-Performance Distributed* Computing (HPDC 13)

[4] Evaluating SOAP for High-Performance Business Applications: Real Trading System, In *Proceedings of the 12th International World Wide Web Conference*.

[5] Dan Davis and Manish Parashar, Latency Performance of SOAP Implementations, Cluster and Computing and the Grid (CCGrid), 2002.

[6] Alex Ng, Shiping Chen, and Paul Greenfield, An Evaluation of Contemporary Commercial SOAP Implementations, *Proc. of the 5th Australasian Workshop on Software and System Architecture*

[7] Kiran Devaram and Daniel Andreson, SOAP Optimization via parameterized client-side caching, *PDCS 2003* (Parallel and Distributed Computing and Systems)

[8] K.Chiu, M.Govindaraju, and R.Bramley, Investigating the Limits of SOAP Performance for Scientific Computing, *HPDC 2002*

[9] N.Abu-Ghazaleh, M.Govindaraju, and M.J.Lewis, Optimizing Performance of Web services with Chunk-Overlaying and Pipelined-Send, *ICIC 2004*

[10] Benkner, S., Brandic, I., Dimitrov, A. et al. Performance of Java Web Services Implementations. In *Proceedings of ICWS '03*. Las Vegas, 2003

[11] Java™ API for XML-based Remote Procedure Call (JAX-RPC) Specification, Version 1.1

[12] R.A.van Engelen and K.A. Gallivan. The gSOAP toolkit for Web services and peer-to-peer computing networks. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002

[13] Satoshi Shirasuna, Hidemoto Nakada, Satoshi Matsuoka, and Satoshi Sekiguchi. Evaluating Web Services based implementations of GridRPC. In *Proceedings of the 11th IEEE International Symposium for High performance distributed computing* (HPDC-11 2002)

[14] Java Performance Tuning, Chapter 4, Jack Shirazi, O'Reilly

[15] Don Box, et al. "Simple Object Access Protocol (SOAP) 1.1" World Wide Web Consortium (W3C)

[16] Erik Christensen, et al. "Web Services Description Language (WSDL) 1.1", World Wide Web Consortium (W3C)

[17] Google Web APIs, http://www.google.com/apis/

[18] Axis Architecture Guide, http://ws.apache.org/axis/

[19] Min Cai, A Comparison of Alternative Encoding Mechanisms for Web Services, *International Conference on Database and Expert Systems Applications (DEXA) 2002*

[20] Fast Infoset. ITU-T Rec. X.891 | ISO/IEC 24824-1

[21] Fast Web Services, Paul Sandoz, et al. August 2003. http://java.sun.com/developer/technicalArticles/Webservices/fastWS/

[22] XML Binary Information Set (XBIS), http://www.xbis.org/

[23] XML Binary Characterization Working Group, http://www.w3.org/XML/Binary