IJICT International Journal of Information & Communication Technology

ITRC

Volume 2- Number 2- July 2010 (pp.43-49)

Scaling Up Performance of Web Services by Pre-Serialization Response Analysis

Parinaz Saadat
Computer engineering department
Iran University of Science and Technology
Tehran, Iran
saadat@comp.iust.ir

Behrouz Minaei-Bidgoli
Computer engineering department
Iran University of Science and Technology
Tehran, Iran
b_minaei@iust.ac.ir

Received: February 06, 2010- Accepted: July 26, 2010

Abstract—Web services are the inseparable part of many scientific applications. Consider communication between two partial differential equation (PDE) solvers on different domains, such applications work by exchanging large arrays containing floating point data with high accuracy. The reliance of web services on SOAP leads to performance degradation in similar scenarios as the serialization of outgoing messages containing large arrays of doubles is a primary SOAP performance bottleneck.

Various aspects of this problem are examined from the point of views including alternative formats, XML coding compression, differential serialization, differential de-serialization, etc. This paper proposes an approach which combines the idea behind differential serialization and differential de-serialization with a completely different implementation.

Keywords- web services; serialization; performance; SOAP

I. INTRODUCTION

Web services have evolved to allow global communication between web applications. The software industry is settling upon SOAP as a common wire format and HTTP as the connection protocol. A client and a Web service communicate using SOAP messages, which encapsulate the in and out parameters as XML. Fortunately, for Web service clients, the proxy class handles the work of mapping parameters to XML elements and then sends the SOAP message over the network [5].

The task of de-serializing the request into objects and serializing response from a common language runtime (CLR) object goes back into a SOAP message hinders SOAP performance.

To illustrate the opportunity for performance enhancement, consider a widely-used web services such as stocks exchange web services, e-banking, and so forth. Such a web service will expect to receive many concurrent requests and the responses to the clients could contain large arrays of floating-point data. The responses to this large amount of concurrent requests share common parts. As a matter of fact serializing only the different portion of responses is the simplest way to avoid any extra work which can be done by a serializer.

In a nutshell, our technique exploits the similarities between responses in order to improve web service response time. It is worthwhile to say that although our approach is to analyze the similarities between responses, we took one step further and utilize the duplicate requests between current calls. This assumption has led to defining an extra phase for analyzing the incoming requests as well.

II. RELATED WORK

The approach in [1] describes the design and implementation of differential serialization, a SOAP optimization technique that can help bypass the serialization step for messages similar to those previously sent by a SOAP client or returned by SOAP based web service. The authors of [2] have introduced the design and implementation of differential serialization's analogue on the server side which is called differential de-serialization (DDS). The idea is to avoid fully de-serializing each message in an incoming stream of messages. Differential de-serialization gets its name because the server-side parser de-serializes the differences between an incoming message and a previous one.

III. THE PROPOSED APPROACH

Our previous work [1] has addressed this problem on the request, by avoiding de-serializing all messages. As the first component which is responsible for handling requests in a client-server model is the web server, it would be the best candidate for handling any pre-processing. As a result we have proposed a middleware which is capable of running on top of any web server. Our middleware looks for exact same messages in any concurrent requests arriving into the web server. In order to be able to discriminate between redundant and non-redundant requests we maintained a trie of incoming parameter sequences. Although the approach showed significant performance enhancement, this improvement largely depends on the probability of receiving requests with completely the same parameters for a service.

Fig 1 illustrates the idea behind both previous and present study.

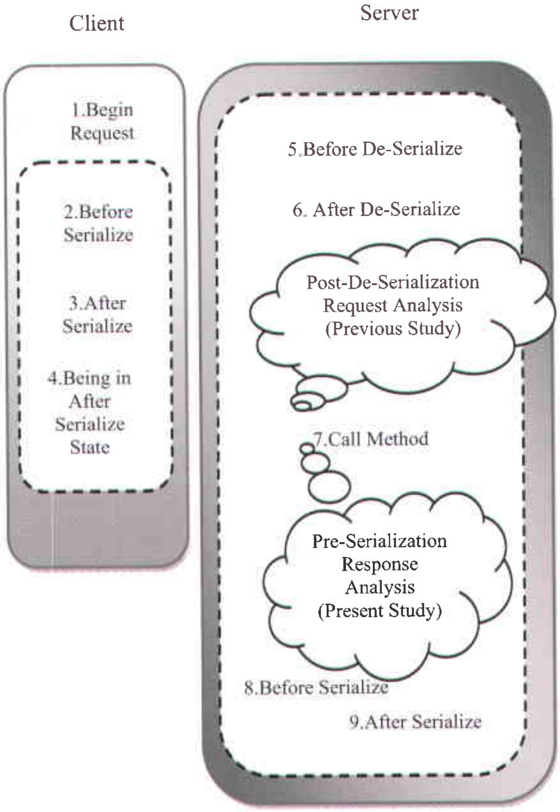


Fig 1. The paradigm of the approach

There is a rather intricate lifecycle to the SOAP message, involving multiple stages of message processing. In this paper the research is focused on the "Before Serialization" stage which occurs immediately after the web method returns and before the return values are serialized and sent back to the client.

As mentioned earlier in this paper, the idea is to take a more general issue into account and take aim at analyzing responses and avoiding performing the expensive stage of serializing large objects after each response is ready.

In order to do so, a means of intercepting SOAP messages in the SOAP message pipeline is needed. The ASP.NET SOAP extension framework represents this feature. Through the SOAP extension architecture one has access to the message as it is de-serialized into objects, and as it is serialized from a CLR object back into a SOAP message. This means that the SOAP message can be reached before and after Serialization/De-serialization process.

IV. IMPLEMENTATION

The process of performance improvement is twofold. The first stage is avoiding processing exact same requests per Current Collection. The second one is avoiding the redundant serialization stage of SOAP responses which share same message portions. As a result we continue this study in two separate sections. The first section describes the methods dealing with requests and the second section focuses on response issues. In order to maintain web service statelessness, we concentrated on "Current Requests" on the web server. So we had to define the term "Current" in this context. In the implementation the term current request is used for incoming messages during a predefined period of time.

Section A discusses the possibility of performance improvement based on requests reaching the web service and discusses the algorithms used for comparing SOAP messages. Section B then describes an optimization based on the responses.

A. Request Post-Deserialization Processing

For post de-serialization request analysis, current requests for incoming messages during a predefined period of time are collected in a dataset which is then passed to the next step, each 2 seconds for further analysis. Next parameters are retrieved from each SOAP request and a parameter sequence is generated for each request. If the sequence of parameters is duplicated, there is no need to do all the job of request processing and serialization of response for every single request.

Request comparison, analysis and processing consist of five main steps, each running in a different thread for maximum performance enhancement. These steps are described in detail in [1] but to fully understand the idea of this phase a pseudo code is also prepared which is shown in Fig 2.

```
R : Queue of all Current Requests
H: New An Empty HashTable
RQ: Init New Queue of Redundant Requests
DQ: Init New Queue of distinct Requests
For each Request in R
{
    P: ExtractCallParameters(R)
    If p is already in H
        Mark Ri as Redundant
        Enqueue Ri To RQ
    Else
        H =AddToHashTable(H,P)
            Enqueue Ri To DQ
}
Send DQ to Web Server for further Processing
Reset timer
```

Fig 2. Pseudo code for pre-processing request

B. *Response Pre-Serialization Processing*

As responses include larger objects to be serialized and sent on wire, the opportunity for performance enhancement by utilizing the proposed technique mostly relies on this phase.

For the reasons mentioned in [1] including maintaining statelessness of web services, the study focuses on Current Responses on the web server. So the term "Current" in this context has to be defined. In the implementation the term *Current Response* is used for messages that are ready to be serialized and are sent over the wire during a predefined period of time. For the purpose of the implementation this predefined period of time was set to 2 milliseconds but can be modified in different situations. A timer is activated and responses are collected and passed to the next step each 2 milliseconds for further analysis where all responses are portioned into chunks of the same size. Each response is chunked each N byte where N is set to 32, 64, 128, 256, 512, 1024, 2048, and 4096.

By now, every message is chunked. The next phase is to compare the corresponding chunks and find identical portions so that all but one of identical portions can bypass the serialization step. Fig 3 illustrates response chunking. As it is shown in the figure responses 1 to 6 are chunked into chunks a to f.

There are many issues concerning chunk size. The larger the chunk size, the more space requirement. On the contrary, smaller chunks will cause more context switch and the process of comparison will become a bottleneck leading to performance degradation.

Different scenarios were simulated to identify the best chunk size.

A thread is initiated for each corresponding chunk in all responses and is responsible for finding common portions in the assigned chunks.

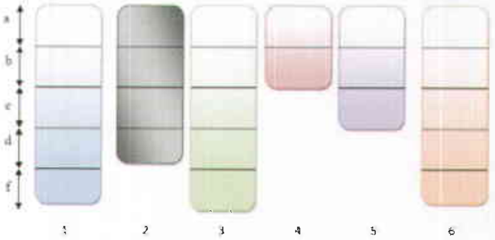


Fig 3. Response chunking

The algorithm used for defining common parts is one of the most challenging issues of this implementation. As it is shown in Fig 4, all corresponding chunks will be assigned to each thread for example chunk a of all 6 responses is assigned to T1, chunk b of all 6 responses to T2 and so on .

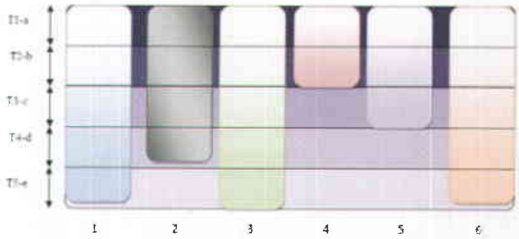


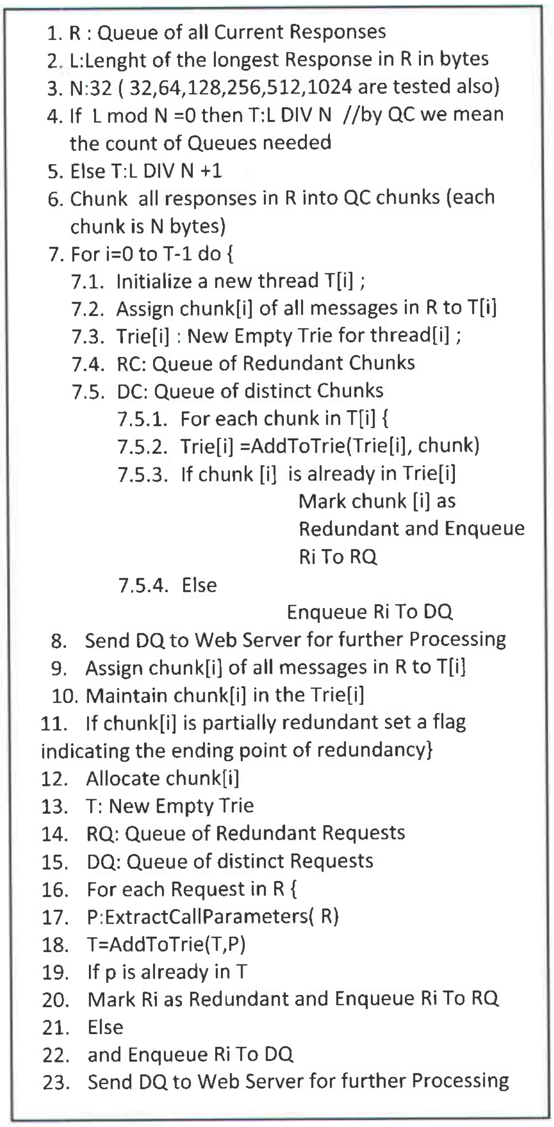
Fig 4. Thread Assignment

The most straightforward algorithm is to simply compare each of n chunks with other n-1 chunk in the collection. In many applications, it is necessary to determine the string similarity. Edit distance approach is a classic method to determine the Field Similarity [8,9]. A well known dynamic programming algorithm is used to calculate edit distance with the time complexity of O(nm). The Hamming distance also can be used.

A faster algorithm had to be chosen otherwise the comparison phase would be a bottleneck itself. The opportunity for performance enhancement depends largely on the finding as much similarities as possible in each chunk. Fine-grained chunk comparison helps the technique perform even better. A trie, or prefix tree, is an ordered tree data structure that is used to store an associative array where the keys are usually strings. All the descendants of a node have a common prefix of the string associated with that node. Through this way by maintaining all chunks in a trie, finding common prefix of any key can be simply determined.

Therefore, each thread maintains its corresponding chunk in a trie. For the example as shown in Fig 4, T1 maintains a trie for chunk "a" for all responses. A key factor for choosing the trie is that for detecting duplicate sequence of length *m,* this prefix tree takes worst case O(m) time. Besides, tries are requiring less space when they contain a large number of short strings, because the keys are not stored explicitly and nodes are shared between keys with common initial subsequences. The granularity of chunks guarantees that this feature can be utilized. Another advantage of using tries over similar data structures is that looking up keys is faster. Looking up a key of length m takes in the worst case, O(m) time.

The pseudo code for accomplishing this step is shown in Fig 5 and 6.

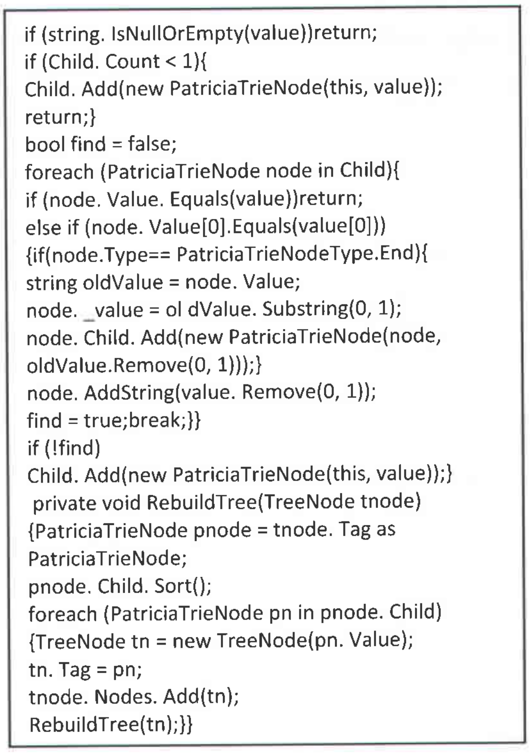


```
1. R : Queue of all Current Responses
2. L:Lenght of the longest Response in R in bytes
3. N:32 ( 32,64,128,256,512,1024 are tested also)
4. If  L mod N =0 then T:L DIV N  //by QC we mean
   the count of Queues needed
5. Else T:L DIV N +1
6. Chunk  all responses in R into QC chunks (each
   chunk is N bytes)
7. For i=0 to T-1 do {
   7.1.  Initialize a new thread T[i] ;
   7.2.  Assign chunk[i] of all messages in R to T[i]
   7.3.  Trie[i] : New Empty Trie for thread[i] ;
   7.4.  RC: Queue of Redundant Chunks
   7.5.  DC: Queue of distinct Chunks
         7.5.1.  For each chunk in T[i] {
         7.5.2.  Trie[i] =AddToTrie(Trie[i], chunk)
         7.5.3.  If chunk [i]  is already in Trie[i]
                          Mark chunk [i] as
                          Redundant and Enqueue
                          Ri To RQ
         7.5.4.  Else
                        Enqueue Ri To DQ
8.   Send DQ to Web Server for further Processing
9.   Assign chunk[i] of all messages in R to T[i]
10.  Maintain chunk[i] in the Trie[i]
11.  If chunk[i] is partially redundant set a flag
indicating the ending point of redundancy}
12.  Allocate chunk[i]
13.  T: New Empty Trie
14.  RQ: Queue of Redundant Requests
15.  DQ: Queue of distinct Requests
16.  For each Request in R {
17.  P:ExtractCallParameters( R)
18.  T=AddToTrie(T,P)
19.  If p is already in T
20.  Mark Ri as Redundant and Enqueue Ri To RQ
21.  Else
22.  and Enqueue Ri To DQ
23.  Send DQ to Web Server for further Processing
```

Fig 5. Pseudo code for post-processing of response

A Binary Search Tree (BST) performs O(log(n)) comparisons of keys, where n is the number of elements in the tree, because the lookups depend on the depth of the tree, which is logarithmic in the number of keys when the tree is balanced. Hence, in the worst case, a BST takes O($m$log(n)) time. Moreover, in the worst case log(n) approaches m. Also, the simple operations tries use during lookup, such as array indexing using a character, are fast on real machines [7].

So far common portions of all corresponding chunks in each thread are identified. It is time to start regular serialization of all responses and send the SOAP message on the wire. To do so, the common portions are serialized once and used for the chunks when it is needed.



```
if (string. IsNullOrEmpty(value))return;
if (Child. Count < 1){
Child. Add(new PatriciaTrieNode(this, value));
return;}
bool find = false;
foreach (PatriciaTrieNode node in Child){
if (node. Value. Equals(value))return;
else if (node. Value[0].Equals(value[0]))
{if(node.Type== PatriciaTrieNodeType.End){
string oldValue = node. Value;
node. _value = ol dValue. Substring(0, 1);
node. Child. Add(new PatriciaTrieNode(node,
oldValue.Remove(0, 1)));}
node. AddString(value. Remove(0, 1));
find = true;break;}}
if (!find)
Child. Add(new PatriciaTrieNode(this, value));}
 private void RebuildTree(TreeNode tnode)
{PatriciaTrieNode pnode = tnode. Tag as
PatriciaTrieNode;
pnode. Child. Sort();
foreach (PatriciaTrieNode pn in pnode. Child)
{TreeNode tn = new TreeNode(pn. Value);
tn. Tag = pn;
tnode. Nodes. Add(tn);
RebuildTree(tn);}}
```

Fig 6. Pseudo code for adding chunks to the Trie

V. EXPERIMENTAL SETUP AND ANALISYS

In this section performance studies to prove that this is a promising approach for improving the overall service call time are discussed in detail. Three subsystems helped us simulate any situation regarding the study:

- Load Simulator

  Generates SOAP request and simulates X Concurrent web service method call per second.

- The Request/Response Analyzer

  Responsible for *post-de-serialization* analysis of *requests* and *pre-serialization* analysis of *responses*.

- A Monitoring tool

  For monitoring the web server's Performance Counters such as Byte Received/Sec, Byte Send/Sec, Total Bytes/Sec, and Connection Attempts/Sec.

All performance tests have been implemented on a single Pentium 4 3.00 GHz machine with 3.24 GB of RAM, and a 100GB SATA Drive.

Multiple situations have been tested so that the results can be compared. To draw a better conclusion, we have implemented each test 10 times, dropped the highest and lowest results, the average of remaining values has been recorded as the overall response time, the factor indicating the performance enhancement or degradation.

The method was the same for all situations, a request to a web service was simulated, and every step was monitored till the response was ready. In each iteration of the test, the *load simulator* retrieved call parameters from a table containing over 12000 parameter sequence of integer, double and string.

The monitoring tool then showed the results of each.

The request post-de-serialization analyzer, running on top of IIS 7.0, read a sequence of "incoming" SOAP messages passes them to the comparison algorithm module where call parameters of SOAP message were retrieved. Only distinct requests could go for further processing by the web server. The effectiveness of this phase on performance largely depends on the amount of exact same requests in each set of concurrent simulations.

The response pre-serialization analyzer then receives the responses for the set of distinct requests. Responses are chunked and assigned to threads.

Table 1 shows the effect of trie depth for different chunk sizes on the serialization times.

Another point is that the depth of trie, parameter sequence length, has also a dramatic effect on performance enhancement.

TABLE 1. THE EFFECT OF TRIE DEPTH FOR DIFFERENT CHUNK SIZES ON SERIALIZATION TIME (MILLISECOND)

	Trie Depth				
Chunk Size (byte)	**15**	**25**	**35**	**65**	**75**
32	2.89	28.7	71.5	144	288
64	0.83	8.24	17.7	42.7	85
128	0.70	6.93	16.0	35.4	71.3
256	0.67	5.67	13.6	28.6	63.9
512	0.57	1.72	4.29	8.03	55.4
1024	0.56	1.32	3.23	5.89	16.7
2048	018	0.75	2.11	3.69	12.5
4096	0.13	0.67	1.85	2.95	7.72

Table 2 lists the overall response time of several simulated scenarios. In this table M indicates message size of response including different array sizes. Each time the responses are chunked into N bytes where N is {32, 128, 256, 512, 1024}. In table 2, negative values shows the situations when the proposed solution caused performance degradation.

TABLE 2. RESPONSE TIME IN MILLISECOND. M INDICATES THE MESSAGE SIZE.

M	Arr. Size	N: Chunk Size (Byte)				
		32	128	256	512	1024
100	25K	-55	-6.8	6.4	6.4	-7.5
	50K	-50	2.1	20.7	27.2	20.4
	75K	-51	1	21.5	30.3	28.2
	100K	-53	0.7	22.2	33.5	33.5
200	25K	-72	-43	-33	-29	-35
	50K	-74	-39	-26	-19	-20
	75K	-73	-40	-26	-18	-17
	100K	-74	-40	-26	-17	-14
500	25K	-80	-59	-51	-47	-49
	50K	-80	-57	-46	-41	-40
	75K	-82	-58	-47	-41	-38
	100K	-83	-58	-47	-40	-37
1000	25K	-86	-70	-61	-55	-51
	50K	-86	-68	-59	-52	-48
	75K	-87	-69	-60	-53	-49
	100K	-88	-69	-59	-53	-48

VI. CONCLUDING REMARKS

This paper has proposed a new middleware for scaling up the web services performance. Web services are largely applied in many scientific applications. There are many web-based data exchange applications that their messages are very similar. For example communication between two partial differential equation solvers on different domains of web, work by exchanging large arrays containing floating point data with high accuracy. Such applications have been applied through scientific simulations and mathematical exchanging operations.

The reliance of web services on SOAP leads to performance degradation in similar scenarios as the serialization of outgoing messages containing large arrays of floating points is a primary SOAP performance bottleneck. The task of de-serializing the request into objects, and serializing response from a common language runtime object back into a SOAP message hinders SOAP performance.

Our proposed middleware runs on the top of web server to take advantage of similar SOAP requests on a web server for a particular web service. Using our approach a large portion of responses can bypass the serialization phase if the message is totally the same.

The plan is to add two supplementary steps to the steps that a SOAP message passes through, during its normal life cycle. These steps include post de-serialization request analysis and pre-serialization response analysis. The former takes advantage of similar SOAP requests on a web server for a particular web service. The latter utilizes the similarities between outgoing messages so that a large portion of responses can bypass the serialization phase before being sent over the wire.

For post de-serialization request analysis, current requests for incoming messages during a predefined period of time are collected into a dataset which is then passed to the next step, (every two miliseconds) for further analysis. Next, parameters are retrieved from each SOAP request and a parameter sequence is generated for each request. If the sequence of parameters is duplicated, there is no need to do all the job of request processing and serialization of response for every single request.

Pre-serialization analysis of responses makes a more affective contribution to the overall performance gain. Before serialization of responses, each *N* bytes of all responses are chunked where *N* is {32, 64, 128, 256, 512, 1024} and a thread is responsible for handling the corresponding chunk in all responses. Then a trie is maintained for the chunks in order to find the common portions in each set of corresponding chunks.

We take aim at analyzing requests and responses so that performing the expensive stage of serializing large objects can be avoided with the help of a middleware running on top of web server. For identifying redundant calls a trie of incoming parameters is maintained for every set of current requests. This way request processing and serialization of the response of same requests will be done only once. In a nutshell, to serialize only the different responses is the simplest way to avoid extra work done by a serializer.

Various aspects of this problem are examined from the points of view of alternative formats, XML coding compression, differential serialization, differential de-serialization, and so forth. We have presented an approach which combines the idea behind differential serialization and differential de-serialization with a completely different implementation.

We leveraged our process management through choosing a trie data structure which is maintained for the sequence collection. Every parameter sequence is to be inserted in the trie. A key factor for choosing trie for detecting duplicate sequence of length *m* takes worst case O(*m*) time, in other words trie structure guarantees that no duplicate parameter sequence is maintained.

After this phase the distinct requests, plus one out of n identical request are deserialized and processed. When the response of that request is ready it is sent to all other identical requests as well.

Our experiments show that our middleware is a promising technique for improving the overall service call time.

It might be worthwhile to notify that although our approach is to utilize the exact repeating portion parameters, a further optimization can be considered. We can enable the middleware so that it can be configured to apply changes made to the result set of response to the serialized responses being maintained in a trie to generate valid results. But this can also lead to a larger percentage of time consumption for the comparison and analysis phase. However, this point will be verified in our future work.

Acknowledgment

We are thankful to Mr. Pedram Zadno Azizi that really helped us in coding and editing the paper. We are also thankful to Mr. Abbas Najafiyan from Computer Research Center of Islamic Sciences, Noor Co., the Noor digital library section, Qom, Iran that helped us in choosing the scope of the experiment for different types of web-based applications. We are also appreciating Dr. Peyvast's help in correcting and editing the paper.

References

[1] B. Minaei, P. Saadat. "SOAP Serialization Performance Enhancement: Design and Implementation of a Middleware". *International Journal of Computer Science and Information Security, Vol.6, No. 1, 2009, PP 105-110.*

[2] N. Abu-Ghazaleh, M. J. Lewis. "Differential Deserialization for Optimized SOAP Performance". 2005 ACM/IEEE conference on Supercomputing, pp. 21-31, Seattle WA, November 2005.

[3] N. Abu-Ghazaleh, M. Govindaraju, and M. J. Lewis. "Optimizing Performance of Web Services with Chunk-Overlaying and Pipelined-Send. Proceedings of the International Conference on Internet Computing (ICIC) ", pages 482–485, June 2004.

[4] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju"Performance of Dynamic Resizing of Message Fields for Differential Serialization of SOAP" Messages. Proceedings of the International Symposium on Web Services and Applications, pages 783–789, June 2004.

[5] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju. "Differential Serialization for Optimized SOAP Performance". Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC-13), pages 55–64, June 2004, Honolulu, Hawaii.

[6] Miranda, Claudio: "Tools and Tips to Diagnose Performance Issues,"The International Conference on JAVA Technology, Zurich. (2008)

[7] K. Chiu and W. Lu. "A Compiler-Based Approach to Schema-Specific Parsing". In First International Workshop on High Performance XML Processing,2004.

[8] David Megginson et al. SAX 2.0.1: "The Simple API for XML. http://www.saxproject.org.

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

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

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

[12] M. R. Head, M. Govindaraju, A. Slominski, P. Liu,N. Abu-Ghazaleh, R. van Engelen, K. Chiu, and M. J. Lewis. "A Benchmark Suite for SOAP-based Communication in Grid Web Services". SC—05: Supercomputing '05, page to appear, Seattle WA, November 2005.

[13] N. Juric,Matjaz and Rozman,Ivan and Brumen,Bustjan and Hericko,Marjan: "Comparison of performance of Web services,WS-Security,RMI and RMI-SSL " The journal of systems and software, 79, 689, 2006.

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

[15] T. Suzumura, T. Takase, and M. Tatsubori. Optimizing Web Services Performance by Differential Deserialization". IEEE/ACM International Conference on Web Services, pages 185–192, Orlando, FL, July 12-15, 2005.

Parinaz Saadat is a postgraduate student from the Department of Computer Engineering, Iran University of Science and Technology (IUST). Thesis title is: "A Middleware for Improving Performance of Web services by Differential Serialization" under supervision of Dr. Behrouz Minaei-Bidgoli. She is currently working as the software analyst and system designer of E-payment systems for Sadad Informatics Corp, the IT department of Bank Melli, Tehran, Iran.

Behrouz Minaei-Bidgoli obtained his Ph.D. degree from Computer Science & Engineering Department of Michigan State University, USA, in the field of Data Mining. Now, he is assistant professor in the Department of Computer Engineering, Iran University of Science & Technology (IUST). He is in advisory board of a Data and Text Mining research group in Computer Research Center of Islamic Sciences, NOOR co. Tehran, Iran, developing large scale NLP and Text Mining projects for Farsi and Arabic languages. He is also managing director of Iran National Foundation of Computer Games in Tehran.