A Consistency-preserving Mechanism for Web Services Response Caching*

Wubin Li\(^1\), Zhuofeng Zhao\(^1\), Kaiyuan Qi\(^1,2\), Jun Fang\(^1\), Weilong Ding\(^1,2\)
\(^1\)Institute of Computing Technology
Chinese Academy of Sciences
P.O.Box 2704, 100190, Beijing, China
\(^2\)College of Information Science and Engineering
Shandong University of Science and Technology
Qingdao Shandong, 266510, China
\{liwubin,zhaozf,qikaiyuan,fangjun,dingweilong\}@software.ict.ac.cn

Abstract

Web services are rapidly emerging as a popular standard technology for sharing data and functionality among heterogeneous systems. Service providers and consumers are loosely coupled and distributed across the network, either within an organization or across organizational boundaries, and therefore, performance becomes a major concern in such a distributed environment. Furthermore, XML is widely used as message format for service providers and consumers in Web services environment. XML message packaging and parsing brings extra overhead to both ends. Web services response latency, as well as throughput, is becoming a bottleneck problem. In this paper, we propose a consistency-preserving mechanism for Web services response caching, which reduces the volume of data transmitted without semantic interpretation of service requests or responses, and accelerates the services response finally. It achieves this reduction through the use of cryptographic hashing to detect similarities with previous results. Experiments with an initial prototype called SigsitAcclerator indicate that our mechanism can lead to significant performance improvement over more straightforward techniques.

1 Introduction

Web services\(^{21}\) are emerging as a standard method of sharing data and functionality among loosely-couple, heterogeneous systems. Many organizations and enterprises are considering exposing their existing data and business logic as Web services (to both internal and external audiences)\(^{7}\). In Web Services environments, service providers and requesters are loosely coupled and distributed across the network, either within an organization or across organizational boundaries. Performance becomes a key factor in such a distributed environment. On the other hand, XML is widely used as message format for service providers and consumers. XML message packaging and parsing brings extra overhead to both ends. Therefore Web services invocation costs more in terms of response time than some other kinds of remote procedure invocation. Reducing the response time of costly remote service invocations in such environments is a critical challenge in many real life cases. Accelerating service response time is a basic problem.

Caching, widely used to improve performance, has been used successfully in many solutions to address the performance issue. And it is also a classic but effective approach for improving the round-trip time for request-response exchanges and for reducing recurring computation in distributed systems. Especially for Web services middleware, cache mechanisms contribute considerably to faster response times and higher throughput\(^{17}\). However, previous can not solve the performance problem and consistency problem simultaneously. Most solutions have generally weakened the consistency restriction or required third-party applications’ support.

The objective of this paper is to propose consistency-preserving mechanism for Web services response caching. Our SigsitAcclerator solution, which is based on this mechanism, attempts to cache Web services response messages

---

\(*This work was supported by the National Basic Research Program of China (973 Program) under the grant No. 2007CB310805, and the National Natural Science Foundation of China under the grant No. 60573117.
at both ends (client side and service side). SigSitAccelerator makes no effort to semantically interpret the contents of requests or their responses. Instead, it relies exclusively on cryptographic hashing to detect similarities with previous results. Hash-based similarity detection has seen increasing use in distributed file systems[12, 18, 19] for improving performance on low-bandwidth networks. However, these techniques have not been used for Web services discipline. Unlike previous approaches that exploits the structure of cache entry[20], SigSitAccelerator treats every soap message as an intact one. And our experiments show that, SigSitAccelerator can improve performance significantly. For Web services that are data-intensive, throughput improvements of up to twofold were observed.

The rest of this paper is organized as follows. Section 2 surveys previous work related to caching Web service responses. Section 3 introduces the proposed mechanism and discusses its details, as well as the design and implementation of the initial prototype called SigSitAccelerator. Section 4 describes the experimental evaluation of our mechanism. Finally, a conclusion is presented in Section 5 plus a list of continuing and future works.

2 Related Work

2.1 Web Service Responses Caching

Distributed object caching is a technique for reducing communication overhead between a server and clients. There is a lot of work on providing remote object caching for distributed systems[1] including CORBA [2] and Java RMI[6], etc. Several issues and corresponding suggestions associated with caching Web services mobile ad-hoc networks were discussed in[8]. Among the several issues that were discussed, two important ones pose main challenges. The first is cache consistency while the second is efficient communication as it relates to performance acceleration.

In[17], an efficient response cache mechanism appropriate for the Web services architecture was proposed. Three optimization methods were advised to improve the performance of our response cache. The first optimization is caching the post-parsing representation instead of the XML message itself. The second is caching application objects. The third optimization is for read-only objects. These methods reduce the overhead of XML processing or object copying. However, consistency problem was not deeply mentioned. Instead, the discussion was based on an assumption that it is the responsibility of the client application administrator to configure a Time-To-Live(TTL) for each operation. And the TTL should be short enough to avoid consistency problems, which is dependent on the services semantics.

In[15], IBM proposed a cache mediation pattern as a reusable solution to accelerate service response in a Service-Oriented Architecture (SOA[13]) environment where messaging middleware is employed as the communication channel. The cache in this pattern resides on the messaging middleware between the service providers and the service consumers. Unlike traditional design patterns, this pattern is more at the infrastructure level and leverages existing capabilities provided by the messaging middleware. Unfortunately, consistency problem was not discussed neither, and therefore, any implementations of this pattern are required to provide specified technics to avoid consistency problem.

2.2 Cryptographic Hash and Applications

The past few years have seen the emergence of many systems that exploit hash-based techniques. At the heart of all these systems is the simple yet elegant idea of identifying contents using the cryptographic hashing of data blocks. The main feature of this approach is semantically neutral, so it can be applied in many different contexts. Successful applications of this idea span a wide range of storage systems. Examples include finding similar files[10], peer-to-peer backup of personal computing files[4], and storage-efficient archiving of data[14].

The same idea can also be used to support data transport over networks with intermittent and highly variable connectivity. The Data-Oriented Transfer(DOT)[5] system breaks the data stream into individual chunks that are addressed by the hash of their content. The sender transmits these hashes to the receiver before transmitting the actual data. Using the hashes, the receiver can retrieve the data from either the original sender or from other sources. In[16], similar principles are applied at the network level. Using synchronized caches at both ends of a network link, duplicated data is replaced by smaller tokens for transmission and then restored at the remote end.

The use of hash-based techniques to reduce the volume of data transmitted has emerged as a common theme of many recent storage systems. These techniques rely on some basic assumptions. Cryptographic hash functions are assumed to be collision-resistant[9]. In other words, it is computationally intractable to find two inputs that hash to the same output. The functions are also assumed to be one-way; that is, finding an input that results in a specific output is computationally infeasible. Menezes et al.[11] provide more details about these assumptions. The above assumptions allow hash-based systems to assume that collisions do not occur. Hence, they are able to treat the hash of a data item as its unique identifier. A collection of data items effectively becomes content-addressable, allowing a small hash to serve as a codeword for a much larger data item in permanent storage or network transmission.

Similarly, SigSitAccelerator’s focus is on efficient trans-
mission of results by discovering similarities with the results of previous request through cryptographic hashing techniques. As some responses can be large-size results, hash-based techniques lend themselves well to the problem of efficiently transferring these large results across bandwidth-constrained links. Furthermore, SigsitAccelerator does not depend critically on any specific hash function. While we currently use SHA-1, replacing it with a different hash function would be simple. There would be no impact on performance as stronger hash functions (e.g., SHA-256) only add a few extra bytes and the generated hashes are still orders of magnitude smaller than the data items they represent.

3 Proposed Mechanism

Our aim is to accelerate the response of Web services invocation at the client side, however, to guarantee consistency, we need to add some trick-like feature at the server side. And we chose agent interposition as the architectural approach to realizing our goal.

3.1 Proxy-Based Caching

As Figure 1 shows, at the client side, SigsitAccelerator in Figure 1(b) is a lightweight code component. Its main function is to mediate communication between the Web services requester side and the remote SigsitAccelerator proxy. It forwards requests from Web services requester, buffers entire results, and responses to Web services requester to acquire results. It provides the ability to retrieve results from compact hash-based descriptions sent by the proxy. To perform this retrieval, SigsitAccelerator maintains an in-memory cache of recently received results.

3.2 Results Handling at both ends

3.2.1 Results Handling at SigsitAccelerator proxy

As Figure 2 shows, the SigsitAccelerator proxy receives response results from Web services provider side, and then check the size of the result. If the size of the result is less than a threshold value (e.g., 200K bytes), it will believe that it is not necessary to generate hash-encodings of that result (it’s too small), but forward it directly to SigsitAccelerator without any more computation. (More details about the threshold value will be discussed in section 3.3.)

Otherwise, the proxy will rapidly execute the hash function to generate hash-encodings of that result. As mentioned before, SigsitAccelerator does not depend critically on any specific hash function. The hash function mentioned here can be SHA-1, SHA-256, SHA-224, SHA-512, etc. When finishing execution of cryptographic hashing, a compact result (hash-encodings of the original result) is generated. Size of the compact result depends on the hash function we used, and is much smaller than the original one. According to the cryptographic hashing theory mentioned in section 2.2, this process can be finished very fast. In our
prototype, we currently use SHA-1 as the hash function, and therefore, the size of each compact result weighs 160 bits.

The next step is to check whether the compact result is in cache. If positive, that means the client has requested for this result before, and then the proxy just needs to transmit the compact result whose size is much smaller than the original one to the client side, and that helps reduce the latency of transmitting data through the internet; otherwise, the compact result is bran-new (never appears before), and then the proxy keep it in the cache, before transmitting the compact result and the original response result to the client side finally.

### 3.2.2 Results Handling at SigsitAcclerator

![Dataflow for results handling at client side](image)

**Figure 3.** Dataflow for results handling at client side

When receiving a result from SigsitAcclerator proxy, SigsitAcclerator examines its data type, as shown in Figure 3. If it’s a compact result, SigsitAcclerator retrieves the intact response result from cache through the use of the compact result as the key; otherwise, the response result will be kept in cache, using the compact result as the key so that it can be retrieved at sometime. When response result retrieved, it will pass to the Web service client. On some occasions as we will discuss in section 3.4, cache miss might happen. But there should be very few misses if the both ends are reliable.

### 3.3 Determine the threshold value(s)

Intuitively, the threshold value should be determined by the bandwidth of the network. For a moderately constrained network, a greater threshold value is preferable since it can reduce the overhead for frequently executing hashing function at the SigsitAcclerator proxy. Nevertheless, for a severely constrained network, it is not advisable to use a great threshold value, because it might cost longer latency when transmitting large-size response results between the client side and the server side.

Furthermore, if there exist more than one clients, for the reason that the bandwidth of each link to the Web services server side might be different, we have two distinct options: the first is to fix an identical threshold value for all the clients, while the second is to use different threshold values for different clients. The question is which is better, how to fix this value, and whether there are any other better solutions.

Besides, relying on the bandwidth of the network, the threshold value can also be dynamically re-adjusted to obtain a better performance. We believe that determining the threshold value(s) would be an interesting direction for the future work. In our SigsitAcclerator prototype, we use an identical threshold value for all the clients and more work will be studied about this issue in the future.

### 3.4 Client Crashing Problem

The discussions mentioned above rely on one basic assumption that both ends of this SigsitAcclerator architecture are reliable enough. In other words, the server side or the client side will not crash. However, once the client side crashed at sometime, we may have to encounter the consistency problem again. For example, client side sends a request message $Msg$ to the server side, and response message $Res$ is cached, using its hash value $V$ as the key. Then unfortunately, the client side crashed, with the subsequence that all entries in the cache would have been cleared after a restart. In this case, if the client side sends the same request message $Msg$ before the result for this request updates in server side, it would fail to retrieve the result from cache through hash value $V$.

We can fix this problem by enabling the SigsitAcclerator proxy caching the response results when caching their hashes. In this way, once cache missing happens, SigsitAcclerator can resend the hash value $V$ to SigsitAcclerator proxy to fetch the result. It is a space-consistency tradeoff which will increase the overhead of SigsitAcclerator proxy but avoid consistency problem. Since both components are under our control, it is relatively simple to achieve this by hard-coding. On the other hand, there might be some better solutions and we will also do more research on this issue in the future.
4 Prototype and Experimental Evaluation

4.1 Prototype and Experimental Setup

The experimental setup consists of two parts: the client side, consisting of our SigsitAccelerator prototype, and the server side, consisting of web services set up by us, and the SigsitAccelerator proxy.

As table 1 shown, our SigsitAccelerator prototype is a multi-threaded system written in Java. For communicating with web services using SOAP, our prototype uses Codehaus XFire tools. Codehaus XFire is a next-generation java SOAP framework. Given a description of a web service in the Web Service Definition Language (WSDL), XFire generates a class such that the web service can be invoked simply by calling a method of the generated class. The input and output types of the web service are also encapsulated in generated classes. When implementing SigsitAccelerator proxy, we use SHA-1 as the cryptographic hashing function. Furthermore, all cache components involved are implemented to only support in-memory cache.

 presente raw text as if you were reading it naturally: 

<table>
<thead>
<tr>
<th>CPU</th>
<th>Platform</th>
<th>Soap Engine</th>
<th>Application Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Pentium 4 3.2 GHZ</td>
<td>Windows XP Professional Edition</td>
<td>Codehaus XFire version 1.2.6</td>
<td>Apache’s Tomcat (version 6.0.14)</td>
</tr>
</tbody>
</table>

Table 1. Experimental Setup Details

At the server side, we use Apache’s Tomcat (Version 6.0.14) as the application server that hosted the Web services which are developed using Codehaus XFire tools. And there is only one web service hosted, which has a table \( T \) (int \( a \), String \( b \), primary key \( a \)) associated with it. Given a value for attribute \( a \), it retrieves the corresponding value for attribute \( b \) from \( T \) (by issuing a SQL query) and returns it. The tables \( T \) are stored using the lightweight MySQL DBMS. Since attribute \( a \) is the primary key, MySQL automatically builds an index on \( a \). Concretely, the data size of \( b \) ranges from 100K to 1M, while \( a \) is supposed to be a positive integer no more than 100.

Figure 4. Experimental Setup

The experimental setup used for performance evaluation can be seen in Figure 4. All machines were 3.2 GHz Pentium 4 (with Hyper-Threading enabled). With the exception of the Web services server, all machines had 1.5 GB of DDR2-SDRAM and ran the Windows XP Professional Edition distribution. The Web services server had 2 GB of DDR2-SDRAM. The machines were connected by a switched gigabit Ethernet network. As shown in Figure 4, the Web service client and Web service server are separated by a network router. This router allowed us to control the bandwidth and latency settings on the network. The bandwidth and latency constraints were only applied to the link between the client and the server for the native case and between the client and the proxy for the SigsitAccelerator case. There is no communication between the client and the Web services server with SigsitAccelerator as all data flows through the proxy.

4.2 Experimental Procedure

4.2.1 Some Assumptions

Based on what we have mentioned in section 3.4 and section 3.3, we fix some assumptions as follows:

- Both sides of this system are reliable and stable. No machine will crash during the experiment.
- We use an identical threshold value for the client, and will not re-adjust it during the experiment.
- The threshold value is set to be 200K bytes.

4.2.2 Experimental Scenarios

In this experiment, we evaluate SigsitAccelerator along two representative scenarios. The first one is on the WAN bandwidth of 5 Mb/s with 50 ms of round-trip latency, representative of severely constrained network paths, while the second one is on bandwidth of 10 Mb/s with 20 ms of round-trip latency, representative of a moderately constrained network path. And we measured results from the two configurations listed as follows:

- **Native**: This configuration corresponds to Figure 1(a). Native avoids SigsitAccelerator’s overhead in using a proxy.
- **SigsitAccelerator**: This configuration corresponds to Figure 1(b). For a given number of clients and bandwidth of network, comparing these results to corresponding Native results gives the performance benefit due to the SigsitAccelerator system.

During each scenario case, 1000 concurrent requests from client side will be sent to the Web services server. Time interval between two nearby requests is less than 3000 milliseconds.
For each request, the client randomly generates a positive integer $a$ which is less than 100, and then sends out the request message to Web services server side to get the corresponding result value $b$ in table $T$ which resides in MySQL database. The latency of response is defined as the time from the delivery moment of the request to the result’s arrival at the client side, which includes the time used for XML parsing. Every request is identified by a request-number, and responses for each request will be traced so that we can compare the performance differences in the Native case and the SigsitAccelerator case. For the sake of presenting the results through figures, we take every 10 requests as a sample, and calculate their average response delay as one request’s.

4.3 Results and Analysis

4.3.1 Trend of Cache Hit Ratio

1000 request parameters are randomly generated, each of which is a positive integer less than 100. So intuitively, if the distributed uniformly, the hit ratio will be increased gradually, as shown in Figure 5.

![Figure 5. Cache Hit Ratio During the Experiment](image)

Higher cache hit ratio is preferable for the SigsitAccelerator architecture, since in that case, according to the dataflow that we discussed in section 3.2.1, more compact results would be transmitted instead of large-size response results, and that will help reduce the overhead of the network, especially when the bandwidth is severely constrained.

The hit ratio depends on two aspects: the first factor is how frequently for the client side to send duplicate requests, while the second is the update frequency of the response result for a request. In our experiment, the response value $b$ for a certain request $a$ is stable, unchanged. However, in real life cases, the situation might be totally different, so we need a benchmarks to evaluate those two factors’ influence over the SigsitAccelerator performance. Unfortunately, there are no available benchmark for Web services response performance testing so far. Consequently, we make the assumption that data in the back-end database would be updated during our experiment. And based on this, we detect how fast that SigsitAccelerator architecture would accelerate the Web services responses.

4.3.2 Severely Constrained Network Scenario

As shown in Figure 6, in the severely constrained network scenario, the average response latency in the native case is 5644.0 milliseconds, while it is 2818.5 milliseconds in the SigsitAccelerator case, and therefore 50.06% acceleration is improved.

![Figure 6. Results Comparison in severely constrained network scenario](image)

According to the severely restriction of network bandwidth, and the data size of the response messages, most latencies for requests are higher than 5000 milliseconds in the Native case. While in the SigsitAccelerator case, the latencies are mostly below 3000 milliseconds, except those anterior requests. The reason why latencies of anterior requests are most likely the same as those are in the Native case, is due to the fact that cache hit ratio maintains low until the client side frequently sends out duplicate requests, as we mentioned before.

4.3.3 Moderately Constrained Network Scenario

Things changes a lot in this scenario, as shown in Figure 7.

In the moderately constrained networks scenario, the average response latency in the native case is 2120.7 milliseconds, while it is 1624.8 milliseconds in the SigsitAccelerator
case. 23.38% is improved, which is not so notable as what is like in severely constrained network scenario. Most latencies for requests are lower than 3000 milliseconds in both cases. The difference of average response delay to fulfill a request in both cases is 495.9 milliseconds, which is so tiny that can be almost ignored in real life cases. Moreover, frequently executing hash computation would increase the overhead of the proxy in this scenario.

In other words, we come to the conclusion that, SigsitAccelerator can increase the response speed by as much as twice for a severely constrained network. Nevertheless, for a moderately constrained network, the impact is not so effective, but still a 23.38% acceleration can be obtained.

5 Conclusions and Future work

Web services have received significant attention and there is a great deal of industry excitement around the opportunities afforded by them. While most of this attention has focused on Web services architectures, the performance problem of Web services response has been ignored to some extent. In this paper, we focus on Web services response latency issue that arises in Web services invocation process. Our solution shows that the impact of WAN accesses to Web service provider can be substantially reduced through the SigsitAccelerator architecture without any compromise of the response message’s strict consistency semantics.

The essence of the SigsitAccelerator architecture is the use of computation at the edges to reduce communication through the Internet. SigsitAccelerator is able to use cryptographic hashes to detect similarity with previous results and send compact hashes of results rather than full results. Our experimental evaluation confirms that SigsitAccelerator, while conceptually simple, can be highly effective in improving throughput and response time. Finally, we believe that several future work will be needed to develop tackle performance problem for Web services:

- Web services cache consistency management protocols help in avoiding consistency problems and in achieving high cache hit ratios. We hope that a standard cache management protocol for Web services will be specified in the near future.

- The SigsitAccelerator prototype is hard-coded in the client side. Hence, taking software modularization, reusability, and portability into consideration, it is an important next step to separate it from application logic.

- In addition, it is also an interesting enhancement to exploit the structure of XML response messages to figure out whether it can yield superior performance improvement.

- Future work mentioned in section 3.3 and section 3.4.

Acknowledgments

We would like to thank Shuying Yan, Lue Hai and many other members of the SIGSIT Laboratory for many valuable discussions and helpful comments. We are also grateful to Ning Li and Mo Yang for their great effort to fix numerous English problems in this paper.

References


