An Aspect-Oriented Approach to Consistency-Preserving Caching and Compression of Web Service Response Messages

Wubin Li, Johan Tordsson, and Erik Elmroth
Department of Computing Science and HPC2N
Umeå University
Umeå, Sweden
{wubin.li, tordsson, elmroth}@cs.umu.se

Abstract—Web Services communicate through XML-encoded messages and suffer from substantial overhead due to verbose encoding of transferred messages and extensive (de)serialization at the end-points. We demonstrate that response caching is an effective approach to reduce Internet latency and server load. Our Tantivy middleware layer reduces the volume of data transmitted without semantic interpretation of service requests or responses and thus improves the service response time. Tantivy achieves this reduction through the combined use of caching of recent responses and data compression techniques to decrease the data representation size. These benefits do not compromise the strict consistency semantics. Tantivy also decreases the overhead of message parsing via storage of application-level data objects rather than XML-representations. Furthermore, we demonstrate how the use of aspect-oriented programming techniques provides modularity and transparency in the implementation. Experimental evaluations based on the WSTest benchmark suite demonstrate that our Tantivy system gives significant performance improvements compared to non-caching techniques.

Keywords- Web Services; response caching; consistency-preservation; hashing; data compression; aspect-oriented programming;

I. INTRODUCTION

Reducing the response time of costly remote service invocations in Web Services environments is a critical challenge in many real life scenarios. Our previous work [1] describes a consistency-preserving mechanism for Web Service response caching that reduces the volume of data transmitted without semantic interpretation of service requests or responses and thus improves the service response time. It achieves this reduction through the use of hashing to detect similarities with previous results. In this paper, we propose an aspect-oriented middleware solution called Tantivy that extends on our previous results. Unlike previous approaches that exploits the structure of cache entry [2], Tantivy treats each SOAP message as a black box and converts it into application-specific data object (henceforth referred to as an application object) before it is cached. Moreover, Tantivy reduces the size of the data that is transmitted between client and services through the use of data compression techniques.

A Web Service framework that supports caching (as opposed to having to add this functionality in each application that invokes the services) is particularly beneficial as this enables the application developer to ignore caching issues [3]. Some earlier research on how to transparently add caching to an application exists [4], [5], but these efforts ignore the consistency aspect. Other solutions provide consistency, but ignore transparency and thus require substantial effort in implementation of caching-enabled applications [6]. In our Tantivy solution, we keep caching and data compression transparent by casting these as aspects and use an AOP framework to add a proxy layer that intercepts Web Service messages.

We evaluate the performance of our solution with the WSTest [7] benchmark suite. Our results show that Tantivy can improve performance significantly, especially for Web Services that communicate with large amounts of data and/or commonly exchange the same messages.

In summary, our contributions are the following:

1) We investigate how textual data compression techniques and optimized cache representation can further improve response time in an existing solution [1] for consistency-preserving Web Service response caching.
2) We demonstrate that caching of Web Service responses can be considered a crosscutting aspect and illustrate how AOP methods can be used to achieve simple, yet flexible, development of out system.
3) We evaluate the performance of our proposed system through industry standard benchmarks and demonstrates that our approach achieves significant performance improvements.

The rest of this paper is organized as follows. Section II describes background information about Web Services, data compression techniques and AOP. Section III introduces the proposed mechanism and describes its overall structure. Section IV describes the implementation of the Tantivy system using AOP techniques and discusses the related application transparency aspects. Section V presents the experimental evaluation. Finally, some conclusions are presented in Section VI followed by a presentation of continued and future work, acknowledgments, and a list of references.
II. BACKGROUND AND RELATED WORK

A. Web Services

In Web Service environments, SOAP provides interoperability between the clients and the service. However, as the client and the hosted services are connected by a network and communicate through XML-encoded messages, substantial overhead is induced due to (de)serialization. Special care must hence be taken to reduce the response latency for Web Service invocations, and thus improve service throughput. In this paper, we adopt caching and data compression techniques to address this performance issue of Web Services invocations.

B. Web Service Response Caching

Research on remote object caching for distributed systems [8] has caught substantial attention, including efforts that target CORBA [9], SOAP objects [10], and Java RMI [11]. An efficient response cache mechanism appropriate for the Web Services architecture is proposed by Takase et al [12]. This mechanism reduces the overhead of XML processing and application object copying by optimized data representation. However, consistency problems are not addressed. Instead, the discussion is based on the assumption that it is the responsibility of the client application to avoid consistency problems by configuring a short enough time-to-live for each operation. It should be remarked that this approach also depends on the service semantics.

The SigsitAccelerator solution [1] tackles the consistency problem through the use of cryptographical hashing [13]. We extend on this approach and add two additional techniques to further improve performance, textual data compression and optimized data representation of cached entries. To avoid implementations of caching features that are hard-coded in the client and the service side, AOP techniques are applied to improve software modularization, reusability, transparency, and portability.

C. XML Compression and SOAP Performance

XML, the foundation of the SOAP protocol, is a self-descriptive textual format for structured data. XML provides a good basis for interoperability and facilitates the adaptation of services, but it is also renowned to be verbose. This verbosity, mainly due to the excessive use of markup and metadata, can cause problems due to communication and processing overhead in resource-constrained environments such as small wireless devices and in environments with network limitations. Fortunately, the impact of this verbosity can be alleviated through the use of text compression techniques. According to a summary [14], three categories of compression algorithms can be used to reduce the verbosity of XML: general-purpose compression agnostic of XML, algorithms based on the general knowledge that the data is XML-based, and techniques that take advantage of the schema used for the particular XML documents to be compressed. Tantivy does not depend on any specific compression algorithm. While we currently use GZip [15], a general-purpose compression algorithm for textual data, replacing this with another compression algorithm is straight-forward.

D. Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a programming methodology for separating crosscutting concerns (behaviors that cut across the typical divisions of responsibility, such as logging and caching) into single units called aspects. An aspect is a modular unit that provides a functionality with crosscutting concern. It encapsulates behaviors that affect multiple classes into reusable modules. Through weaving rules specified by the developer, aspects are incorporated to form the final system. As a result, a single aspect can contribute to the implementation of multiple methods, modules, or classes, and can thus increase both the reusability and the maintainability of the code.

Figure 1. Conceptual illustration of how weaving rules incorporate a caching aspect into an application to make it caching-enabled.

Figure 1 shows the basic principle of adding caching transparently to an application through aspect weaving. AOP allows us to dynamically modify the static model to include the code required to fulfill the secondary requirements without having to modify the original model. Better still, we can often keep this additional code in a single location rather than having to scatter it across the existing model, as we would have to if we were using object-oriented techniques only. There is a considerable number of technologies that support the AOP paradigm. We choose AspectJ [16] for Tantivy as it is among the most mature and full-featured frameworks available today.

III. THE TANTIVY SOLUTION

A. Main Principles of Tantivy

The two aspects of Web Service communication overhead are data representation and data transfer. Each of these provides an opportunity to reduce communication overhead. In the context of Web Services, data representation requires the transformation of application data into internal representations in the form of XML Infosets [17]. The type of
representation determines the amount of data that has to be transmitted. One additional transformation required is that from the internal representation of what should be sent, to something that actually can be sent. For Web Services, this means that the internal XML Infoset representation is serialized into an XML document before it is transferred over the network.

As Web Services are platform neutral and thus cannot depend on a specific wire protocol, it is not possible to affect the data transfer step. This leaves us with only one opportunity for communication performance improvements, namely data representation. In this paper, we adopt two optimization methods to improve the performance of Web Service responses. The first improvement is caching of the post-parsing representation (application object) instead of the XML message itself in the client side. The second optimization method is to reduce the size of the data representation transmitted over the network through caching and data compression techniques.

### B. Proxy-Based Caching

To guarantee consistency, a caching layer is added to the Web Service framework. This layer is provided through AOP techniques instead of hard-coded implementations in the service engine. The Tantivy Client shown in Figure 2(b) is a lightweight component that mediates communication between the Web Services client and the remote Tantivy Proxy. It forwards requests from the Web Services client, buffers the entire results, and responses to the Web Services client to acquire the results. The Tantivy Client provides the ability to retrieve results from hash-based descriptions (digest) sent by the proxy by maintaining an in-memory cache of recently received results.

![Image](image.png)

**Figure 2. Conceptual overview of the Tantivy architecture as compared to a native Web Service.**

The Tantivy Proxy shown in Figure 2(b) does not examine any request messages received from the Tantivy Client but directly forwards them to the Web Service. Instead, the proxy is responsible for inspecting response results received from the Web Service provider. The proxy rapidly generates hash-based encodings of the results and caches these encodings. If the results are similar to previous ones, only the hash digests are sent to Tantivy. Note that the proxy does not need to keep the actual response messages but only the digests. This enables the proxy to scale well also when many clients are using the same service.

### C. Results Handling at Service and Client Side

Figure 3 shows the dataflow for results handling at the Tantivy Proxy side. The Tantivy Proxy first receives response results from the Web Services provider, and then checks the size of the result. If the size of the result is less than a threshold value (e.g. 200 KB), the proxy does not generate a hash digest of that result, but forwards it directly to Tantivy Client.

![Image](image.png)

**Figure 3. Dataflow for results handling at the Tantivy Proxy side.**

Otherwise, the proxy generates a digest of the result. Tantivy does not depend on any specific hash function. Modern hash functions [13] computes hash digests very fast. The size of the digest depends on the hash function used, but is in general much smaller than the size of the original response. In our prototype, we currently use SHA-1 [18] as the hash function and the size of each hashed result is thus 160 bits.

The next step is to check whether the hashed result already is stored in the cache. If so, the client has requested this result before and the proxy only needs to transmit the hashed result. Otherwise, the hashed result is new and the Tantivy Proxy stores it in the cache. The proxy also compresses the original response message to a compact one before finally transmitting it to the client side. This way, large messages are always compressed and the amount of data transmitted over the network is reduced even if cache misses occur at the Tantivy Proxy side.

Figure 4 shows the overall dataflow in the Tantivy Client. The first step in the client is to inspect the type of a result received from the Tantivy Proxy. If the result message is a hash digest, Tantivy retrieves the stored response result from cache through the use of the received hash digest as key.
Otherwise, Tantivy checks whether the result is compressed and if so, the result is decompressed to the original one. Next, the response result is stored in cache with the hash digest as key before it is finally passed to the Web Service client.

At the client side, the data representation for cached data is made efficient by deserializing responses only once and storing the resulting application objects in the cache. This way, upon a cache-hit, the client can immediately fetch the application object from cache without any parsing or deserialization process, and the response latency is further reduced. In detail, before delivering a response message to the client, the response result is converted to an application object in advance. This process is fulfilled by an XML parser, which can be based either on DOM [19] or SAX [20]. If it is a DOM parser, a DOM tree object, as the post-parsing representation, is created from the XML message. If the parser is a SAX parser, the SAX parser reads the XML documents and notifies the deserializer of the SAX events sequentially. The deserializer constructs the application objects from the DOM tree object or the SAX events sequence. As the parsing and deserialization of XML messages constitutes a large part of the Web Services overhead, caching of application objects instead of XML objects can significantly improve the performance of service response caching.

IV. DESIGN AND IMPLEMENTATION

Aspect-oriented programming provides an elegant approach to perform caching by treating it as concern that cuts across the application. We choose Codehaus XFire [21] as the Web Service framework to build Tantivy on and add a caching aspect as a new feature. XFire, which has been merged with the Celtix project [22] to constitute Apache’s CXF incubation project [23], is a Java SOAP framework. It facilitates service-oriented development through its easy to use API and support for standards such as WS-I [24]. We have also made similar experiments with other frameworks such as Axis [25], Axis2 [26] and JBossWS [27], and we believe that all of these frameworks could be used to implement Tantivy through AOP techniques.

We choose AspectJ as the AOP framework. AspectJ is a seamless aspect-oriented extension to the Java programming language that enables clean modularization of crosscutting concerns. The AspectJ language exposes a set of join points that are well-defined places in the execution of a Java program flow.

```java
public aspect Tantivy{
    //Crosscutting actions for cryptographic hashing
    pointcut hashing(MessageContext context):
        Object msg=context.getDataContextProperty
            { PostInvocationHandler.RESPONSE_VALUE }
            if( msg.size() > Threshold )
            { //Compute the hash key for the message.
                Object key=Cryptographic_Hashing(msg);
                if( IsCacheHit(key) )
                { //Only need to send the key.
                    msg->key;
                }
                else{
                    //Save message in the cache.
                    Save2Cache(key,msg);
                    //Compress message to compact data.
                    Object compactData=
                        Data_Compression(msg);
                        msg.compactData;
                }
            }
        //Message size is too small,
        //no need to handle it.
    }
}
```

Figure 5. Code example that outlines the caching aspect in the Tantivy Proxy.

Figure 5 shows the implementation of the Tantivy Proxy through a pointcut (a piece of code that can be inserted at a join point) and advice declaration in the AspectJ language. This code follows the dataflow process described in Figure 3 (a). The code includes a pointcut called `hashing` that defines the execution of the `sendMessage` method in the `ServiceInvocationHandler` class (or any subclass) that takes a first argument of type `MessageContext` (lines 3-7 in Figure 5). This pseudocode also defines an advice that executes posterior to the specified pointcut (the `sendMessage` method at Line 10).

The client side of the Tantivy system (see Figure 3 (b)) is similar. Note that the pointcuts and advices that define the weaving rules to be applied are specified as entities separate from the individual aspect modules. The weaving of the final system from individual aspects is performed by the AspectJ compiler, *ajc*.
V. EXPERIMENTAL EVALUATION

A. Web Service Benchmarking

The WSTest [7] benchmark suite is used to evaluate the performance of the Tantivy framework. These benchmarks are developed by Sun Microsystems and later extended by Microsoft. WSTest is designed to measure the performance of various types of Web Services calls and includes the following benchmarks:

- **EchoVoid** - sends and receives an empty message. This benchmark performs neither serialization nor deserialization.
- **EchoStruct** - receives an array of arbitrary length as input parameter and returns this array. The structures in the array contain one element each of type integer, float, and string. The longer the array, the more work is required in deserialization and re-serialization of the SOAP object to and from XML.
- **EchoList** - sends and receives a linked list of any size, where each element in the list consists of the same structure as used in EchoStruct.
- **EchoSynthetic** - sends and receives a structure that contains a byte array of varying length.
- **GetOrder** - simulates a request for a complete purchase order for an e-commerce service. This benchmark takes three integer input parameters and returns an order object. The order object is a complex structure that includes order header information, a customer structure with shipping address and billing address structures, as well as any number of objects.

WSTest (version 1.5) consists of a multi-threaded application that performs multiple Web Service calls in parallel in order to simulate a real life scenario with multiple clients that access the services. To avoid the overhead of other platform components, the Web Service operations perform no business logic but simply return the input parameters. WSTest measures the throughput of a system handling multiple types of Web Service invocations. The notion of a Web Service invocation here corresponds to one request-response cycle. WSTest reports the throughput (average number of Web Service invocations executed per second) and the response time (average time it takes to process a request). These metrics are reported separately for each of the five operations.

B. Experimental Setup

The experimental setup consists of a client side extended with a Tantivy Client module. This client has five threads, one for each benchmark. The other part in the setup is the server side that implements the WSTest services. The server side is extended with a Tantivy Proxy module. The two sides have identical system configurations, shown in Table I.

The client side and the services are connected by a network router that allows us to control the bandwidth and latency settings on the network. We focus our evaluation on three network configurations; 5 Mb/s, representative for severely constrained network paths, 20 Mb/s, representative for moderately constrained network paths, and 100 Mb/s, representative for unconstrained networks. The last setup is used to investigate any potential overhead of Tantivy in situations where bandwidth is not a limiting factor.

Each client submits a mix of invocations, with 20% of the calls for each of the five benchmarks. After a warmup period of 300 seconds, each client thread initiates invocations of the benchmark services as per the defined mix. A new invocation is started as soon as the prior one is completed. The number of invocations executed and the response time is accumulated during a steady state period of 600 seconds and is reported at the end of the execution. Moreover, invocations during each execution have a certain repetition rate. For example, in our second experiment, 8% of the invocations are duplicate. For repeating invocations the same request parameters are used and the response results from the server are thus the same. Steering this repetition rate, i.e., the cache-hit ratio, enables us to study the performance impact of caching in the Tantivy system.

Using this setup, we measured results for various combinations of number of clients, cache-hit ratio, and network bandwidth for the following two configurations:

- The Native configuration, corresponding to Figure 2(a) where no proxy is used.
- The Tantivy configuration, corresponding to Figure 2(b) where the Tantivy layer is used. For a given number of client threads and a certain network bandwidth, comparing these results to the corresponding Native ones investigates the potential performance improvements.

C. Performance Analysis

Figure 6 presents the average number of requests served per second and the average response time for these requests in the scenario with 5 Mb/s bandwidth. There are no duplicate requests during this test run and the cache-hit ratio is thus zero. Due to the computation overhead for hashing and data compression, Tantivy is a bit slower than Native for the EchoVoid, EchoStruct, and EchoSynthetic benchmarks. For the EchoList and EchoOrder tests, both with larger amounts of data being sent, the use of compression in Tantivy results in slightly better performance than Native despite the 0% cache-hit ratio.

When the cache-hit ratio is increased to 8%, we observe in Figure 7 that the benefits of caching balances out the
overhead induced Tantivy and the performance Native and Tantivy is almost identical for the EchoVoid, EchoStruct, and EchoSynthetic benchmarks. Furthermore, for this configuration, Tantivy gives around half the response time of Native for EchoList and EchoOrder. Increasing the cache-hit ration even further to 13 %, we see additional improvements for the EchoList and EchoOrder benchmarks, with Tantivy being almost five times faster than Native for the previous benchmark. As Figure 8 shows, at this higher cache-hit ratio Tantivy results in a significant performance improvement over Native also for the EchoStruct benchmark.

We observe in Figure 9 that for 8 % cache-hit ratio and a 20 Mb/s network, the response times of Native are similar to those of Tantivy, except for the EchoList benchmark, where Tantivy performs substantially better. As caching and data compression are more beneficial for slower networks, we note that the performance improvement of using Tantivy is much higher for 5 Mb/s networks than for 20 Mb/s ones.

As illustrated in Figure 10, Tantivy gives only minor performance improvements over Native for a network with a bandwidth of 100 Mb/s. We thus conclude there is no significant advantage in using Tantivy in a 100 Mb/s network for services that exchange messages with sizes representative for the WSTest experiments. However, we foresee that for very large message sizes, the compression capabilities in Tantivy can improve performance also for high bandwidth networks.
Comparing the relative impact of cache-hit ratio and network bandwidth on Tantivy performance, we observe that the setup with a 5 Mb/s network and a cache-hit ratio of 13% is faster than one with a 20Mb/s network and 8% cache-hit ratio and in fact very close to the performance of Tantivy for a 100 Mb/s network and 8% cache-hit ratio. We thus conclude that the main performance improvement of the Tantivy system is achieved for services where the message exchange patterns result in high cache-hit ratios, whereas the network bandwidth is a minor factor for Tantivy performance.

Figure 11. Throughput and average response time with 8 % cache-hit ratio and a bandwidth of 20 Mb/s and different number of concurrent client threads.

Figure 11 illustrates, for a varying number of client threads, the performance of Native and Tantivy for a 20 Mb/s network and a cache-hit ratio of 8 %. We note that the performance improvement of Tantivy over Native increases with the number of client threads. This suggests that Tantivy is a scalable solution that can improve both response time and throughput for highly loaded services.

VI. CONCLUSIONS AND FUTURE WORK

Web Services have received substantial attention and there is a great deal of industry excitement around the opportunities they provide. Most of the attention today has focused on Web Services architectures, leaving the performance problem of Web Service responses. With applications based on Web Services, site-to-site traffic volume grows significantly, which can be a problem for performance sensitive applications. In this paper, we focus on the response latency issue that arises in Web Services invocations. Our solution demonstrates that the impact of low network performance can be substantially reduced through caching and compression without any compromise to the strict consistency semantics of service response messages.

The essence of our Tantivy architecture is the use of computation at the edges of the Internet to reduce communication overhead in the network. Tantivy uses hashing to detect similarity with previous results and sends either hashes of results or compressed results rather than original results. Our experimental evaluation based with the WSTest benchmark confirms that Tantivy, while conceptually simple, can be highly effective in improving service throughput and response time.

To the best of our knowledge, Tantivy is the first system that combines the use of hash-based techniques, application object data representation, and data compression techniques with caching of response messages to improve Web Service performance. Finally, we foresee several future directions to tackle the performance problem for Web Service responses:

- Web Service cache consistency management protocols would help in avoiding consistency problems and in achieving higher cache-hit ratios.
- The performance of the Tantivy middleware system relies on some parameters, e.g., the compression threshold value mentioned in Section III-C, which must be configured before the system is started. Preferably, these parameters should be adjusted according to the current network conditions. Work on self-tuning approaches may be a solution to address this issue.
- Another interesting topic would be to investigate if the herein proposed technique with SOAP message response caching could be applied also to commonly occurring subelements of a SOAP message, instead of only to whole messages. This would allow further performance improvements for services that frequently exchange a common set of data objects, e.g., shopping basket contents for e-commerce services.

ACKNOWLEDGMENTS

We thank Zhuofeng Zhao, Jun Fang, and the other members of the SIGSIT Laboratory for their contributions to the previous prototype of this work. We are also grateful to the anonymous reviewers for their constructive comments. Financial support is provided by the European Community’s Seventh Framework Programme ([FP7/2001-2013]) under grant agreement no. 215605 (RESERVOIR).

REFERENCES


