An Optimized Strategy for Data Service Response with Template-Based Caching and Compression

Zhang Peng[1,2], Xu Kefu[1,2,*], Li Yan[3], and Guo Li[1,2]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2] National Engineering Laboratory for Information Security Technologies, Beijing, China
[3] National Computer Network Emergency Response Technical Team, Beijing, China
zhangpeng@software.ict.ac.cn, xukefu@iie.ac.cn

Abstract. Data service is a specialization of Web service, and end-users can synthesize cross-organizational data by composing data services. As composite schemes overlap each other, some primitive data services could be called repeatedly by composite data services, so that the response delay and server load are aggravated. In this paper, an optimized strategy for data service response with template-based caching and compression is proposed. Firstly, the strategy uses the message template to extract the application-relevant values from SOAP messages. Secondly, the strategy holds the objects from application-relevant values rather than XML representations to decrease the overhead of SOAP message parsing. Thirdly, the strategy uses the XMill compression algorithm to decrease the volume of data transmitted. Extensive experiments based on Spring-WS-Test benchmark demonstrate the strategy is an effective approach to reduce response latency and server load compared to non-caching tehcnniques.

Keywords: data integration, message template, data caching, data compression, SOAP, data service.

1 Introduction

Data services are software components that providing rich metadata, expressive languages, and APIs for service consumers to use to send queries and receive data from service providers [1,2]. Data service hides the complexity of the multi-source and heterogeneous data sources, and helps the implementation of user-steering data integration. Compared to the traditional data integration techniques, data service separates the data access interface from the information system, which not only achieves loose coupling between integrated schemes and data sources, but also has good scalability. More importantly, data services have the same data schema, so that the user could directly integrate data souces without middleware schema. Through data service composition, the scattered data from the organizations can be integrated seamlessly to respond to transient business needs [3,4]. Nonetheless, as a specialization of Web services, the very feature that makes data service universally usable for structured and

* Corresponding author.

A.R. Lomuscio et al. (Eds.): ICSOC 2013 Workshops, LNCS 8377, pp. 394–405, 2014.

semi-structured data sources, namely the adoption of the ubiquitous XML standard, makes it difficult to reach the performance level required by large-scale data access. A major performance bottleneck resides in widespread SOAP message processing. The reason for SOAP message processing performance criticality is twofold:

1. On one hand, SOAP communication produces considerable network traffic, and causes high latency.
2. On the other hand, and perhaps more importantly, the generation and parsing of SOAP message, and their conversion to and from in memory application data can be computationally expensive.

In addition, there are special features for data service as shown in Table 1 to differentiate the effect providing service to make the SOAP processing performance become more criticality.

Table 1. Data Providing Service vs Effect Providing Service

Aspects	Data Providing Service	Effect Providing Service
Core Function	Data Query	Business Process
Transfer Type	Information Data	Effect Data
Data Volumn	Large-scale	Small-scale
Intertal Logic	Data Access Logic	Business Logic

We found that most SOAP messages have similar byte representations. SOAP messages created by the same implementation have the same message structure. In addition, the number of SOAP implementations is relatively small. Therefore it seems that the number of message patterns with which data service processor has to deal should be very small. This is especially more likely in enterprise data integration, where the message patterns are very limited and might be known to the server beforehand. In such cases, it is inefficient to repeatedly parse the almost-same messages each time.

```
<? xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:xsd="…"
xmlns:SOAP-ENV="…"
xmlns:xsi="…">
<SOAP-ENV:Body>
<nsl:getReport xmlns:nsl="…">
<nsl:eventReport>XXX</nsl:eventReport>
</nsl:getReport>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
<? xml version="1.0" encoding="UTF-8" ?>
<soap-Envelope  xmlns:xsi="…"
xmlns:xsd ="…"
xmlns:soap="…">
<soap-Body>
<getReport xmlns="…">
<eventReport>XXX</eventReport>
</getReport>
</soap-Body>
</soap-Envelope>
```

Fig. 1. SOAP messages created with Apache Axis and .NET

We also found that the application has far less interest in the literal message structure than the application-defined data structure. Figure 1 show the SOAP message examples emitted by Apache Axis and .NET respectively. For both messages, the application focuses on the string "XXX" rather than on the associated namespace prefixes, etc. In other words, what is important for the application is returning the event information corresponding to the symbol XXX, but not the parsing of the entire SOAP envelope.

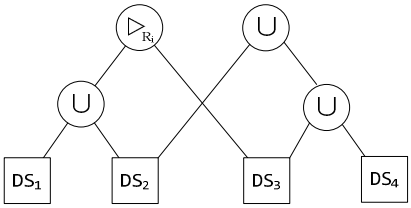


Fig. 2. The data services are called repeatedly

In addition, as composite schemes overlap each other, some primitive data services usually are called repeatedly by different composite data services [2]. For example, one client creates two composite data services denoted by (DS1∪DS2) ▷ Ri DS3 and (DS2∪DS3)∪DS4 as shown in Figure 2, where DS1, DS2, DS3 and DS4 are primitive data services, ∪ is union operation and ▷ Ri is join operation with respect to attribute Ri. When the two composite data services are executed, the DS2 and DS3 will be called repeatedly, and the SOAP transfer involves the some results. So how to optimize the data service communication based on above facts is our goal.

In this paper, a middleware Data Service Accelerator (DSA) is designed. The middleware takes into account the caching data structure and reduces the volume of data transmitted through the combined use of caching of recent responses and data compression techniques. We keep caching and data compression transparent and add a proxy layer that intercepts exchange messages. The paper is organized as follows: Section 2 introduces some related works. Section 3 introduces the DSA with the template-based caching and compression in details. Section 4 is experiment and discussion. Section 5 sums up with several concluding remarks.

2 Related Works

In Data 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 data service invocations, and thus improve service throughput. In this paper, we adopt template-based caching and data compression techniques to address this performance issue of data services invocations.

A. Template-based

As mentioned previously, SOAP processing performance enhancement has been widely researched [5, 6, 7]. Many approaches build on the simple observation that SOAP message exchange usually involves a number of highly similar messages. Several proposals addressing SOAP performance enhancement exploit the differential SOAP parsing, in order to gain in performance, e.g., reducing execution time, increasing throughput, and saving on network traffic. The main idea is to identify the common parts of SOAP messages, to be processed once, regardless of the number of messages. The main approaches to differential SOAP parsing include Template-based [5], Multiple Templates [6] and Detecting Repeatable Structures [7].

B. Response Caching

Research on remote object caching for distributed systems [8] has caught substantial attention, including efforts that target CORBA, SOAP objects, and Java RMI. An efficient response cache mechanism appropriate for the Web Services architecture is proposed by Takase et al [9]. This mechanism reduces the overhead of XML processing and application object copying by optimized data representation. We extend on this approach and add two additional techniques to further improve performance, textual data compression and optimized data representation of cached entries.

C. XML Compression

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 [10], 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. DSA does not depend on any specific compression algorithm. While we currently use XMill [11], a general-purpose compression algorithm for textual data, replacing this with another compression algorithm is straight-forward.

One can realize that the above techniques are not mutually exclusive, but are rather complementary. Unfortunately, interference and synergy between different techniques is not yet completely understood, and less works make some preliminary research about combing these techniques. In this context, this paper makes efforts toward combining these techniques, and gains some benefits.

3 Template-Based Caching and Compression

Based on the observations above, we implemented a middleware Data Service Accelerator to execute three tasks. The first is to use the message template to extract the application-relevant values from the SOAP messages. The second is to hold the objects from the values rather than XML representations to decrease the overhead of XML parsing. The third is to uses the XMill compression algorithm to decrease the volume of XML transmitted. Here we describe its design and implementation.

3.1 Template-Based Extraction

Figure 3 shows an overview of our template-based processor, where the solid and dotted lines denote the processing flow and the data flow respectively.

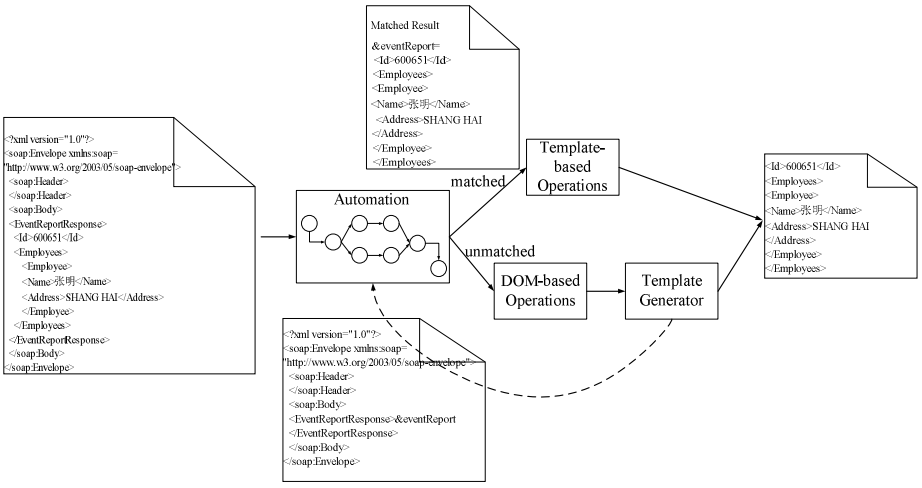


Fig. 3. The application-relevant values extraction based on message template

First, the incoming message is matched against the automaton that contains multiple message templates in a merged form. If the message matches any template, the application-relevant values are extracted using the template and DOM processing is done based on the extracted values. Otherwise, the processing is performed by an ordinary DOM-based processor and a new template corresponding to the unmatched message is generated. The new template is merged into the automaton, and thereafter any messages having the same message structure can be matched against the template. The resulting messages are same in both cases. The message template describes the byte-level structure of SOAP messages. It consists of a few kinds of message fragments: constant fragments corresponding to the unchanging parts in the messages and variable fragments corresponding to the variable parts. All the application-relevant values are regarded as variable parts. In Figure 3 the variable fragments are denoted

as "${variable_name}" and printed in bold italic. Since the template-based processor initially has no information about the XML tree structure without help from DOM-based processor, message templates are generated in cooperation with the DOM-based processor. In order to tell where the application-relevant values start and end, we implemented a specialized XML parser that can record the offset and length information for each XML node. Using the DOM created by the specialized parser, the DOM-based processor can select all of the application-relevant nodes and specify them as the variable fragments with the offset and length information. Finally, the message parts that were not selected in the previous process are consolidated as a constant fragment, and a message template is generated from these fragments. Note that DOM-based operations are performed only when a message with a new structure arrives. Based on our observations, most messages match the existing templates and the frequency of creating templates is low.

Let's introduce the automaton. The automaton provides two interfaces, one for matching the incoming messages and one for learning the new message templates. If the incoming message matches any of the existing message templates, the matching result returned is a map object that contains pairs of variable names and the corresponding actual variable values. In Figure 3, for example, the incoming message is matched against the message template and a map of {eventReport, <id>600651</id> <Empolyees>…</Empolyees>} and so on is returned. The DOM operations are done based on the extracted values. However, if the matching failed, the processor would have had to create a new message template and make the automaton learn that template, in preparation for the later incoming messages for which the new template will be applied. Learning message templates means updating the state diagram in the automaton. All of the templates inside the automaton are stored in a merged form and can be represented as a state diagram.

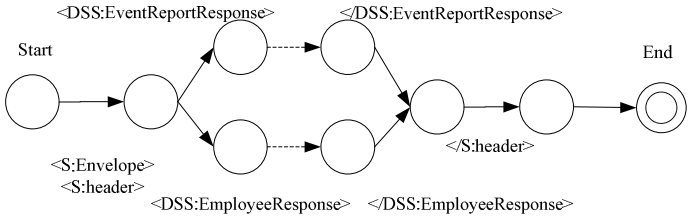


Fig. 4. The message template

Figure 4 shows a simple example of a state diagram which has been created from two message templates, where a solid line and a dotted line denote an actual state transition and one or more transitions (intermediate states might be omitted), respectively. Concretely in this example, this means two message templates have already been learned. Since "<S: Envelope><S: Header>" and "</S: Header> ..." are common to both messages, their paths are merged. As seen in the figure, one XML node (or tag) does not necessarily correspond to one state.

After the application-relevant values have been extracted from the message, there are no substantial differences in the operations themselves between the

template-based processor and the DOM-based one. They differ in the way they retrieve the values from the message. The template-based processor uses template matching while the DOM-based one traverses DOM tree. The next step is how to transfer the extracted values over the network. For this purpose, the caching and compression techniques in DSA are given in following sections.

3.2 Proxy-Based Caching

In the context of Web services, data representation requires the transformation of application data into internal representations in the form of XML Infosets[1]. For data services, this means that the internal XML Infoset representation is serialized into an XML document before it is transferred over the network. As data services are platform neutral and thus cannot depend on a specific wire protocol. This leaves us with only one opportunity for communication performance improvements, namely data representation. For this purpose, a caching layer is added to the data service framework. This layer is provided through interface techniques instead of hard-coded implementations in the data service engine. The DSA Client shown in Figure 5 is a lightweight component that mediates communication between the data services client and the remote DSA Proxy. It forwards requests from the data services client, buffers the entire results, and responses to the data services client to acquire the results. The DSA 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.

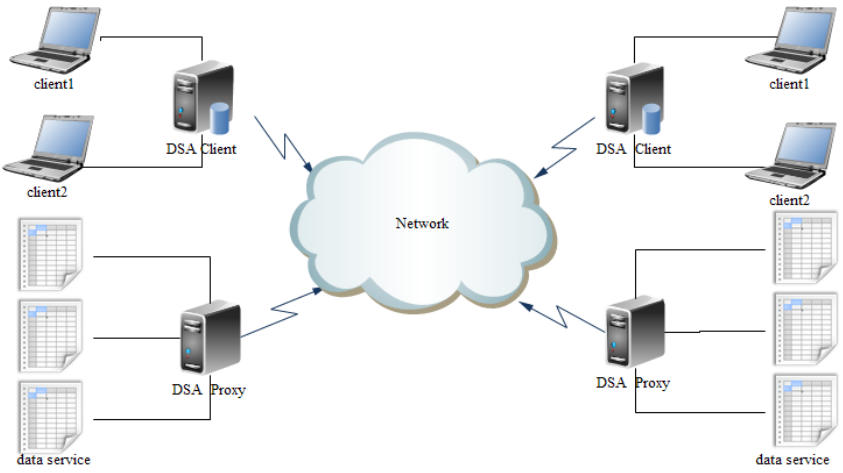


Fig. 5. Conceptual overview of the DSA architecture

[1] http://www.w3.org/TR/xml-infoset/

The DSA Proxy shown in Figure 5 does not examine any request messages received from the DSA Client but directly forwards them to the Data Service. Instead, the proxy is responsible for inspecting response results received from the Data 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 DSA. 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.

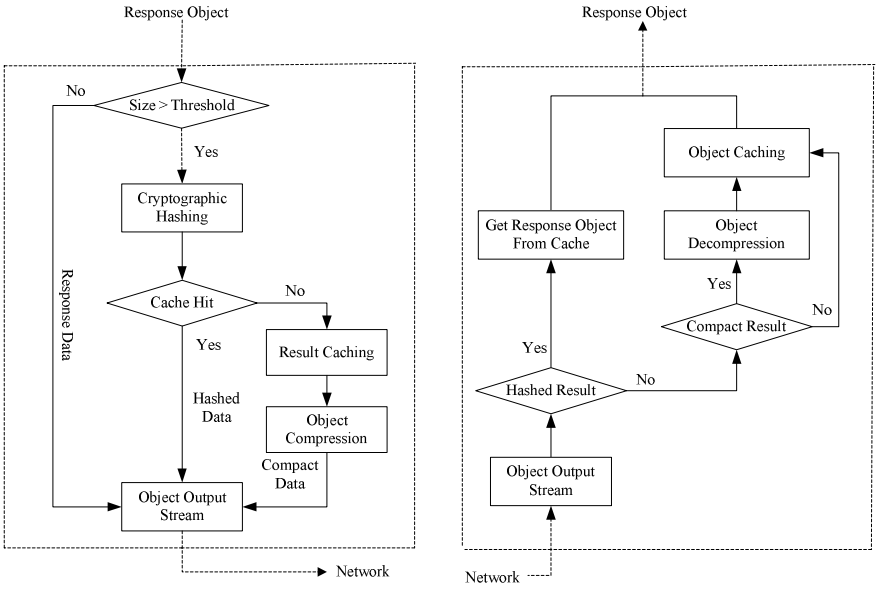
3.3 Data Handling at Service and Client Side

Figure 6(a) shows the dataflow for results handling at the DSA Proxy side. The DSA Proxy first receives response results from the Data Services provider, and then checks the size of the result. If the size of the result is less than a threshold value, the proxy does not generate a hash digest of that result, but forwards it directly to DSA Client. Otherwise, the proxy generates a digest of the result. DSA does not depend on any specific hash function. Modern hash functions 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[12] 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 DSA 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.

Figure 6(b) shows the overall dataflow in the DSA Client. The first step in the client is to inspect the type of a result received from the DSA Proxy. If the result message is a hash digest, DSA retrieves the stored response result from cache through the use of the received hash digest as key. Otherwise, DSA 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 passed to the DSA client.

At the client side, the data representation for cached data is made efficient by de-serializing responses only once and storing the resulting objects in the cache. This way, upon a cache-hit, the client can immediately fetch the object from cache without any parsing or de-serialization 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 object in advance. This process is fulfilled by an XML parser, which can be based either on DOM or SAX. 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 de-serializer of the SAX events sequentially. The de-serializer constructs the objects from the DOM tree object or the SAX events sequence. As the parsing and de-serialization of XML

messages constitutes a large part of the Data Services overhead, caching of objects instead of XML objects can significantly improve the performance of service response caching.



(a). Dataflow for data handling at the DSA Proxy side.

(b). Dataflow for data handling at DSA Client side.

Fig. 6. Dataflow for data handling

Experiment and Analysis

3.4 Benchmarking

We test the performance through industry standard benchmarks from Spring-WS-Test[2]. Spring-WS-Test consists of a multi-threaded application that could performs multiple data service calls in parallel in order to simulate a real life scenario with multiple clients that access the services. Spring-WS-Test measures the throughput of a system handling multiple types of data service invocations.

3.5 Experimental Setup

The experimental setup consists of a client side extended with a DSA Client module. This client has five threads, one for each benchmark. The other part in the setup is the server side that implements the data services. The server side is extended with a DSA

[2] `http://javacrumbs.net/spring-ws-test/`

Proxy module. The two sides have identical system configurations, of which CPU Intel Core i5, 3.09 GHz, Main Memory 4 GB, Operating System Win XP Professional Edition, Application Server Apache Tomcat, version 6.0.14. The client side and the server side 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 DSA 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. 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 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 DSA 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 5 where DSA layer is not used.
- The DSA configuration, corresponding to Figure 5 where the DSA 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.

3.6 Performance Analysis

When the cache-hit ratio is increased to 8 %, we observe in Figure 7 that the benefits of caching balances out the overhead induced DSA and the performance Native and DSA is almost identical for the EchoVoid, EchoStruct, and EchoSynthetic

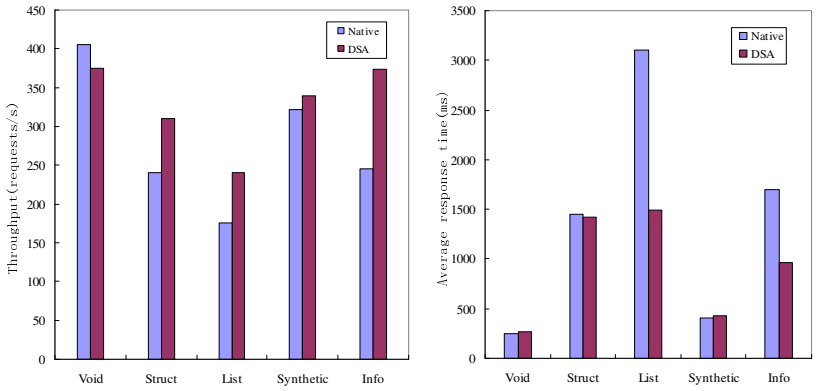


Fig. 7. Throughput and response time with 8 % cache-hit ratio and a bandwidth of 5 Mb/s

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

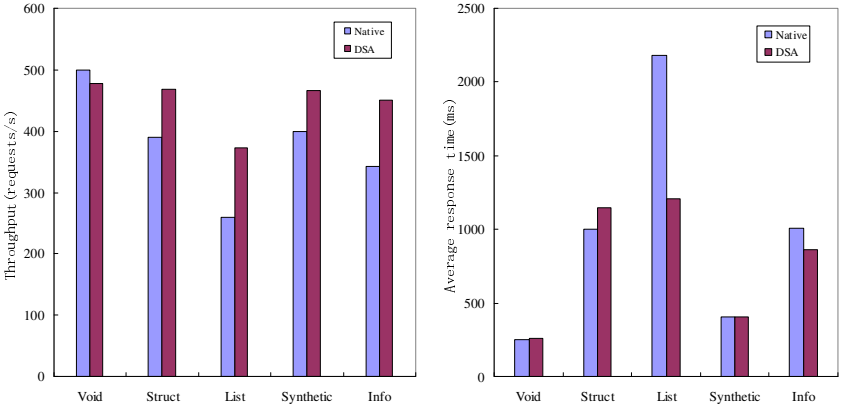


Fig. 8. Throughput and response time with 8 % cache-hit ratio and a bandwidth of 20 Mb/s

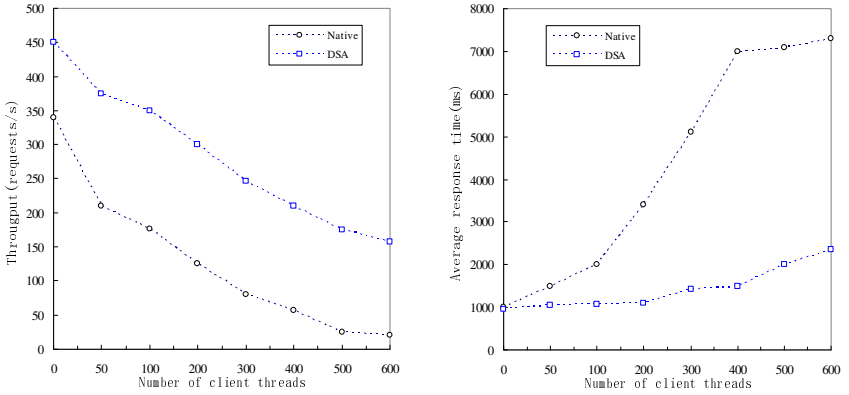


Fig. 9. Throughput and response time with 8 % cache-hit ratio and a bandwidth of 20 Mb/s and different number of concurrent client threads

Figure 9 illustrates, for a varying number of client's threads, the performance of Native and DSA for a 20 Mb/s network and a cache-hit ratio of 8 %. We note that the performance improvement of DSA over Native increases with the number of client threads. This suggests that DSA is a scalable solution that can improve both response time and throughput for highly loaded data services.

4 Conclusion

Data 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 data service modeling and composition, leaving the performance problem of data services somewhat ignored. In this paper, we focus on the response latency issue that arises in data services invocations. Our solution demonstrates that the impact of low network performance can be substantially reduced through caching and compression.

Acknowledgements. The research work is supported by Supported by the National High Technology Research and Development Program 863 under Grant No.2011AA010703; the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant No.XDA06030602; the China Postdoctoral Science Foundation under Grant No. 2013M541076.

References

1. Carey, M.J., Onose, N., Petropoulos, M.: Data Services. Communication of ACM 55(6), 86–97 (2012)
2. Zhang, P., Wang, G., Ji, G., Liu, C.: Optimization Update for Data Composition View Based on Data Service. Chinese Journal of Computers 34(12), 2344–2354 (2011)
3. Han, Y., Wang, G., Ji, G., Zhang, P.: Situational data integration with data services and nested table. In: Service Oriented Computing and Application, pp. 1–22 (2012)
4. Lin, H., Zhang, C., Zhang, P.: An optimization strategy for mashups performance based on relational algebra. In: Sheng, Q.Z., Wang, G., Jensen, C.S., Xu, G. (eds.) APWeb 2012. LNCS, vol. 7235, pp. 366–375. Springer, Heidelberg (2012)
5. Takeuchi, Y., Okamoto, T., Yokoyama, K., Matsuda, S.: A Differential-Analysis Approach for Improving SOAP Processing Performance. In: Proceedings of the IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE 2005), pp. 472–479 (2005)
6. Makino, S., Tatsubori, M., Tamura, K., Nakamura, Y.: Improving WS Security Performance with a Template Based Approach. In: ICWS 2005, pp. 581–588 (2005)
7. Teraguchi, M., Makino, S., Ueno, K., Chung, H.-V.: Optimized Web Services Security Performance with Differential Parsing. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 277–288. Springer, Heidelberg (2006)
8. Bal, H.E., Bhoedjang, R., Hofman, R., et al.: Performance evaluation of the Orca shared-object system. ACM Trans. Comput. Syst. 16(1), 1–40 (1998)
9. Takase, T., Tatsubori, M.: Efficient Web Services response caching by selecting optimal data representation. In: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 2004), pp. 188–197. IEEE Computer Society (2004)
10. Ericsson, M.: The effects of XML compression on SOAP performance. WWW Journal 10(3), 279–307 (2007)
11. Liefke, H., Suciu, D.: XMill: an efficient compressor for XML data. ACM SIGMOD Record, 153–164 (2000)
12. National Institute of Standards and Technology. Secure hash standard (May 2010), `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf`