Efficient Web Services Response Caching by Selecting Optimal Data Representation

Toshiro Takase Michiaki Tatsubori
IBM Research, Tokyo Research Laboratory
E30809@jp.ibm.com / mich@trl.ibm.com

Abstract

This paper discusses the design for an efficient response cache mechanism appropriate for the Web services architecture. The important feature of Web services is its interoperability between heterogeneous platforms. This interoperability is based on widely accepted standards such as XML, SOAP, and WSDL. We describe a response cache mechanism for Web services client middleware without any extensions to these standards so that the client can participate transparently in the existing Web services community. We propose three optimization methods 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. For this optimization, we show some copying processes that are dependent on the type of cached objects. The third optimization is for read-only objects. These methods reduce the overhead of XML processing or object copying. We have implemented a prototype of a response cache on Apache-Axis, and evaluated these optimization methods through experiments for Google Web services. Finally, based on the experimental results, we discuss the optimal configuration of these methods based on data types.

1. Introduction

Currently, the Web services are spreading widely throughout the world. Web services are an enabling technology for interoperability within a distributed, loosely coupled, and heterogeneous computing environment. However, previous studies have shown that the performances of Web services are relatively poor because of the encoding processing [4]. Therefore, technologies to improve the performance of Web services are needed. In this paper, we discuss the design of an efficient response cache appropriate for the Web services architecture.

One of the use cases for Web services assumed in this paper is a portal site scenario. When an end user accesses the portal site, an HTML page is returned from the site to the browser. Assume that the portal site uses several back-end services, such as stock quote services, search services, and news services, for example. These services are built using Web services, and the portal site sends requests to the servers of companies that provide these services. These services are provided using XML [16] messaging, the SOAP [17] message layer protocol, and WSDL [18] as a description of the interfaces. Such inter-enterprise data exchanges using XML messages as the "on-wire" data representation are called Web services. Once a service application is developed as a Web service, the interface description can be easily published to external partner companies and the service can easily be provided to new partner companies.

Caching is 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. The main design considerations of these caching systems are what and where to cache, and how to manage the cached data. In order to reduce communication overhead, client-side response caching is effective. Client-side caching can potentially achieve the greatest reduction in communication overhead since in the best cases actual communication can be avoided with this approach. While this technique has been used in Web browsers and proxies, it is also effective for a Web services client.

This paper focuses on the enhancement of the response cache mechanism for Web services client middleware. Client-side caching stores cached data in the client middleware. Response caching holds responses (the results of requests) from services so that the cached results can be reused for the same requests instead of processing the same steps again. In Web services the reduction in cache processing overhead on client machines is an important consideration. Communication overhead does not affect the overall service throughput anymore with effective client-side caching. Instead, the processing overhead greatly affects the throughput. This paper addresses ways to reduce the overhead of caching in Web services client.

The rest of this paper is structured as follows: First, we survey related work in Section 2. Section 3 discusses design issues for Web services response caching in client middleware. In Section 4, we describe our implementation of client cache middleware that caches application objects whenever possible to reduce the overhead of the data processing required for Web services. Section 5 shows the

Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)
1063-6927/04 $20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

results of experiments measuring each kind of overhead for various forms of cached data and for various methods of processing the stored data to copy the application objects to be returned as service results. We discuss the configurations of the cache mechanisms for Web services client middleware in Section 6 and conclude the paper in Section 7.

2. Related work

Since caching is a traditional approach for accelerating request-response processing, many researchers have made valuable explorations into response caching. Our research focused on removing the caching overhead introduced by Web services technology.

2.1. Remote object caching

Remote object caching is a technique for reducing communication overhead between a server and clients. In this approach, the data originally existing at the server-side is replicated at the client itself (or on other servers closer to the client) so that the client can request the service from a nearby server with lower communication costs. There is a lot of work on providing remote object caching for distributed systems [3] including CORBA [15] and Java RMI [11], just to mention a few. However, in order to support remote object caching techniques in the Web services world, agreements about the replication of server-side objects are required between the clients and servers. Moreover, the replication of server-side objects cannot be used for most inter-enterprise distributed systems. Caching server-objects from another company is often impossible for various reasons such as business strategy or security policies. For example, a client company using a market analysis service company may not be allowed to store a complete copy of the original survey data even if the client company has permission to access the statistical results of the survey. Caching the response data may be possible even in such a case.

2.2. Web page caching

Response caching for Web pages (HTML documents) has often been used in Web browsers and Web proxies. Web page access can be considered as a sort of call-by-copy operation. A Web proxy like [5, 7, 12] caches Web resources in its cache table using URLs as the keys. Since a Web proxy should pass a fetched and stored resource as is to its client through a network, there is no need to process the content of a resource when returning the response using a cached value or to make a copy of the resource. On the other hand, Web services client caching can reduce the cached data processing overhead because response data must be converted to application objects from XML documents. We discuss the optimization methods for cached data representation in detail in Section 3.3.

Web page caching in a Web browser has the chance of applying the overhead reduction technique proposed in this paper. A Web browser could cache GUI components rendered from an HTML document instead of caching the document. This reduces the response time by skipping the process of parsing HTML and rendering GUI components, at the cost of using storage space. However, the overhead of GUI component construction from HTML is small for human users of a Web browser. It is unlikely that an unexpected heavy load would concentrate at an end-user's Web browser, so the overhead removal often does not pay for the memory consumption.

Cache consistency management for Web page caching has been studied by many researchers [1, 5, 7, 12]. For example, studies on caching at Web proxies have examined server-driven cache consistency management based on leases [13]. Dynamic Web data caching at the server-side [6, 14] has often been related to server-driven cache consistency management. Yin et al. surveyed a range of server-driven consistency protocols that had been proposed by many researchers and they reported on the performance impact of these protocols in a sports event Web site [2]. There also exists a combination of client-pull and server-push [9]. We believe that these cache consistency management protocols can be applied to Web services middleware and these technologies are orthogonal to our optimization regarding the cached data representation. In Section 3.2, we discuss cache consistency.

2.3. DB query result caching

A general problem with caching database query results is determining which queries are affected by changes that occur to the source database. This is a well-researched problem [10, 8] and we do not make any contributions to this field, which is related to cache consistency.

In a naively implemented response cache system for Web services, the cost of returning cached data could be relatively large compared to cache systems for flat data such as the result of a database query. This is because the process of constructing application objects from data cached as the result of a service request can be heavy for the Web services middleware if the cached data are presented as XML messages. The overhead of XML processing causes negative effects on the throughput of services and on the response time of the services.

3. Design considerations

In this section, we discuss design considerations specific to Web services client caching, but the discussion in

this section is not limited to any specific middleware implementation. We consider three aspects of Web services caching, (1) parameter passing semantics, (2) cache policy and consistency, and (3) cached data representation. With respect to the first and the second aspects, we take interoperability into consideration as the first priority. Regarding the third aspect, we optimize processing as much as possible, because the optimization here in the middleware is transparent to the client and server applications and does not break the interoperability.

3.1. Parameter passing semantics

The model of Web services is not a shared object model like CORBA, but a request-response model. The parameter passing semantics supported by Web services are call-by-copy only. Call-by-reference semantics have some advantages over call-by-copy semantics. In particular, passing relatively large data may often be more efficient using call-by-reference semantics. However, we do not extend the parameter passing semantics of Web services because the extension requires change of existing service implementations.

In call-by-copy semantics, a returned client-side object and the original server-side object have different identities. Even if the client application modifies the response data, the original data stored at the server should not be modified. The identity of the response data is not shared between the client application and the server application.

In the same way, application objects should not be shared between the client application and the cache of the client middleware. The application objects in the cache should be the response at that time when the client invoked the service. If the cache returns only the reference value of the application objects, the application objects are shared between the client application and the cache. When the application objects are shared, the cached objects reflect changes to the application objects by the client application. Then at the next cache hit, the cached object modified by the client application can be returned.

In order to avoid side effects from changes by the client application, the application objects should be copied into the cache. The copy is required at the time of a cache hit and at the time when the response application objects from the server are stored into the cache. The copy method should implement a deep copy. The deep copy here means copying all objects in the object tree. If the application object is not copied deeply, the internal downstream data in the object tree may be shared. This copy is not required in caching XML messages, because the application object returned to the client application is constructed by the deserializer. Even if the client application modifies the application object, the cached XML message is not changed. At the time of a cache hit, new application objects are constructed every time.

3.2. Cache policy and consistency

Cache policy here means which operations are cacheable and which are uncacheable. Actually, the policy should be determined by the server originating the response. However, we suggest that these cache policies are configured by a client application administrator or deployer, because there is no widely accepted standard for the exchange of Web service cache policies at this time.

We assume that the client application administrator should know server application semantics to some degree. This configuration of the cache policy should be the responsibility of the client-side administrator. Basically, some kinds of retrieval operation are cacheable, but some kinds of update operation are uncacheable.

Table 1. Operations in Google/Amazon Web services

Google Web services	doSpellingSuggestion, doGetCachedPage, doGoogleSearch
Amazon Web services	KeywordSearch, TextStreamSearch, PowerSearch, BrowseNodeSearch, AsinSearch, BlendedSearch, UpcSearch, SkuSearch, AuthorSearch, ArtistSearch, ActorSearch, ManufacturerSearch, DirectorSearch, ListManiaSearch, WishlistSearch, ExchangeSearch, MarketplaceSearch, SellerProfileSearch, SellerSearch, SimilaritySearch
	GetShoppingCart, ClearShoppingCart, AddShoppingCartItems, RemoveShoppingCartItems, ModifyShoppingCartItems, GetTransactionDetails

Table 1 lists all operations included in the Google [19] and Amazon [20] Web services. For example, all the three operations in Google Web services are cacheable. One possible cache policy configuration for Amazon Web services is that 20 search operations listed in the upper part are cacheable and the 6 “shopping cart” operations listed in the lower part are uncacheable.

For cacheable operations, we have to determine how long a cache entry for a response to the operation is valid. This is cache consistency problem. We believe this consistency policy should be also determined by the server originating the response. However, in keeping with our commitment to interoperability, we assume that it is the responsibility of the client application administrator to configure a Time-To-Live (TTL) for each operation. The TTL should be short enough to avoid consistency problems, which is dependent on the service’s semantics. For example, it is reasonable that one hour is short enough for operations in Google Web services. In unpredictable heavy load situations, even a relatively short TTL can be enough to achieve a large cache-hit ratio. In an extreme example, response caching, which reduces the processing overhead, is effective against denial of service (DoS) attacks that send the same requests repeatedly.

IEEE COMPUTER SOCIETY

Cache consistency management protocols help in avoiding consistency problems and in achieving high cache-hit ratios. As described in Section 2.2, many cache consistency management protocols have been proposed. Using these technologies for Web services client middleware seems to be possible. Addressing cache consistency for Web services is also interesting but is beyond the scope of this paper since it is not standard and, if these protocols are adapted, the client and server are bound tightly. We hope that a standard cache management protocol for Web services will be specified in the near future.

In HTTP 1.1, a standard method of cache consistency management is provided. Although Web services are independent of transport protocols like HTTP, in many cases, HTTP is used. In HTTP caching, the consistency is checked in accord with HTTP headers like Cache-Control and If-Modified-Since. A Cache-Control header can describe whether or not the response is cacheable and can specify its expiration time. The If-Modified-Since header enables conditional requests and then a server can return an empty response with status code 304 (Not Modified) if the response has not been updated. Since the reduction of caching overhead discussed in this paper is orthogonal to cache consistency management, this mechanism in HTTP can be applied to our response caching in Web services.

In this paper, we describe the response cache in Web services middleware. This response cache can be used without any changes to the user client application running on the middleware. The application developer does not have to include cache functionality in the application logic during development. Although a cache function can be hard-coded in the user client application, meta-functions like caching should be separated from the application logic. If we take software modularization, reusability, and portability into consideration, this separation is desirable. Instead of hard-code in application logic, we suggest that the policy of the cache can be adaptively configured by an administrator or deployer of the client application.

3.3. Cached data representation

In this subsection, we discuss cached data representations for performance optimization. Figure 1 shows the architecture of a response cache in Web services client middleware. In Web services, the user client applications exchange application-specific data objects with the server applications. In this paper, these application-specific data objects will be simply referred to as “application objects”.

Requests to and responses from Web services are XML messages on the wire. First a request application object is serialized to an XML message. Then, the XML message is sent to a server application. A response XML message from the server application is parsed by an XML parser. If the parser 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 SAX events sequence.

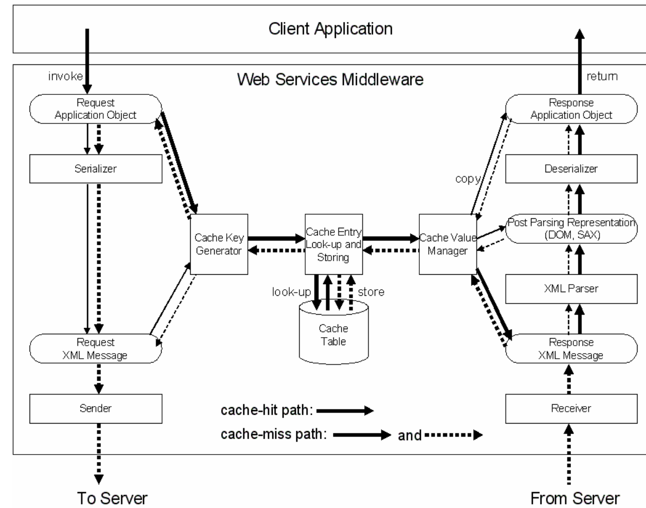

Figure 1. The architecture of a response cache in Web services client middleware

The response cache creates cache keys from requests and stores the responses as cache values. That is to say, a cache entry consists of a pair of a cache key and cache value. For a later equivalent request, the same cache key is generated and the cache table is searched for the cache key, and then the stored cache value is returned as if the response came from the server application.

In Web page caching, the cache key is the request URL. On the other hand, in Web services, because the requests are XML messages, we can generate the cache key from the request XML message itself. However, the serialization of the XML messages has a high overhead. We describe cache key generation from application objects in more detail in Section 4.1. For now, we only note that it is generated from the endpoint URL, operation name, and all parameter names and values.

Because the responses from Web services are XML messages, we can store the response XML message itself as the cache value. However, the parsing and deserialization of XML messages has a very high overhead. We propose three optimization methods in order to improve the performance of our response cache. The first optimization is caching post-parsing representations such as DOM objects or SAX events sequences instead of the XML messages themselves. Using this optimization, XML parsing can be avoided. The second is caching application objects. As described in Section 3.1, when caching application objects, the application object must be copied deeply. The third is optimization for read-only objects. As a special case, read-only objects like immutable objects do not need

IEEE COMPUTER SOCIETY

to be copied. Since the values cannot be modified, read-only objects have no side effects even if the objects are shared between the client application and the cache.

In Figure 1, the thick solid arrows show an example processing path for a cache hit. The thick dashed arrows show additional processing paths for cache misses. In this example, the cache key is generated from the request application object and the response XML message is cached as cache value.

4. Implementation

In this section, we discuss our prototype implementation of the client response caching on Apache-Axis [21], one of the most popular Web services middleware systems. Apache-Axis is an open source implementation in Java. First we describe cache key generation. Then we describe our optimization of the cache value data representation.

4.1. Cache key generation

We evaluated several methods to generate cache keys from request data. Table 2 briefly summarizes these methods and their limitations. In the following parts of this subsection, we discuss the advantages and disadvantages of each method in detail.

Table 2. Cache key data representation

Cache key data representation	Key generating method	Limitation
XML message	Not required	None
Application object	Java serialization mechanism	Serializable object
	toString method	Object which has toString method

4.1.1. Generation from XML messages. In Web services, the messages that travel on the network between client applications and server applications are encoded using XML. In this method, the cache key is generated from the request XML messages. However, when this method is used, it is necessary to perform serialization each time, even for cache hits. Since the overhead for serialization is relatively high, the performance of this method is not very good. Since the XML message is the message exchanged between the client application and the server application, we can apply this method to any type of application object.

4.1.2. Generation from application objects. In order to avoid the serialization process for cache hits, we show below some methods for generating some part of the cache keys from the application objects of the parameter values. The complete cache key is generated from the concatenation of the endpoint URL, operation name, and parameter names and values.

A. Java serialization mechanism

In this method, the Java-serialized form of the application objects of the parameter values are used as a part of the cache key. In Java serialization, the target object and all of its fields are serialized into a byte array, except for transient fields. The objects referenced by the fields are also serialized recursively. Therefore the Java-serialized form is adequate as a cache key. To apply this method, all of the objects in the application object tree must be serializable.

B. toString method

In this method, the results of the toString method for the application objects of the parameter values are used as a part of the cache key. If the application object does not have its own toString method, the toString method of the java.lang.Object class is extended. The toString method of the Object class is based on the memory address, so it is not suitable a as cache key. To apply this method, the application objects must have their own appropriate toString methods.

4.2. Cache value data representation

We evaluated several methods to store response data into the cache. Table 3 briefly summarizes these methods and their limitations. In the following parts of this subsection, we discuss the advantages and disadvantages of each method in detail.

Table 3. Cache value data representation

Cache value data representation	Copying method	Limitation
XML message	Not required	None
SAX events sequence	Not required	None
Application object	Java serialization mechanism	Serializable object
	Copying by reflection API	Bean object, Array object, etc.
	Copying by clone method	Cloneable object
	None (Passing by references)	Read-only object, Immutable object

4.2.1. Caching XML messages themselves. In Web services, the messages that travel on the network between client applications and server applications are encoded using XML. In this method, the XML messages are cached. At the time of a cache hit, the new application object is created from the cached XML message. Even if the created object is modified by the client application, the

cached XML message is not modified. Therefore, there are no side effects. However, when this method is used, it is necessary to perform XML parsing each time. Since the overhead for XML parsing is very high, the performance of this method is not good. Since the cached XML message is the message exchanged between the client application and the server application, we can apply this method to any type of application object.

4.2.2. Caching SAX events sequences. Apache-Axis uses the SAX parser as its XML parser. The SAX parser reads XML documents and notifies the deserializer of SAX events. Therefore, we modified our cache so that it could record the SAX events sequences. Table 4 shows an example of a SAX events sequence. At the time of a cache hit, the cache replays the recorded SAX events sequence again. In this case, although the process of replaying a SAX events sequence and the deserialization are required, the XML parsing is not required. As with the method of caching using the XML messages, since the new application object is created in deserializer, there are no side effects, and we can apply this method to any type of application object.

Table 4. An example of a SAX events sequence

XML document	SAX events sequence
<?xml version="1.0"?> <doc> <para>Hello, world!</para> </doc>	start document start element: doc start element: para characters: Hello, world! end element: para end element: doc end document

4.2.3. Caching application objects. Here, we describe some methods for caching application objects. As described in the preceding section, in caching application objects, we must beware of possible side effects. In order to avoid side effects, the application objects should be deep copied into the cache. We consider three methods for deep copy, Java serialization, reflection copy, and clone copy. These methods have different limitations and we cannot apply these methods to all types of application objects.

A. Copying using the Java serialization mechanism

In this method, the Java-serialized form of the application object is stored into the cache. In Java serialization, the target object and all of its fields are serialized into a byte array, except for transient fields. Also, the objects referred to by its fields are serialized recursively. In Java deserialization, another new object of same type and having the same values is reconstructed from the serialized byte array. That is to say, we can copy the object deeply by using Java serialization and deserialization. In this case, in storing the application object into the cache, the application object is serialized. At the time of a cache hit, the application object is deserialized. The overhead at the time of a cache hit is the overhead for Java deserialization.

Java serialization cannot be done to all types of objects. If the target object does not implement java.io.Serializable interface, the object cannot be serialized. Also, all of the objects referred by its fields must be recursively serializable. Since the application object may be an application-specific object, the object may not necessarily implement the serializable interface. This Java serialization method can be applied only to an application object that is deeply serializable. In the Java serialization process, if an object that is not serializable exists in the target object tree, then an exception is thrown by run-time system. Therefore, the middleware can automatically detect whether or not the application object is serializable.

B. Copying by using the reflection API

We developed a deep copy method by using the Java reflection API. The method can deeply copy bean-type and array-type objects. The term bean-type here means a type that has a default constructor and public setter methods and getter methods for all of the fields that should be serialized.

For bean-type objects, the method creates another new instance and sets the original value of all of its fields to the new instance. If there are fields that refer to a mutable object, another new instance of the mutable objects is created recursively. Immutable objects do not have to be copied (See 4.2.4.). For array-type objects, the method creates another new array and copies the original value of all of the items to the new array. If the type of the item object is a mutable type, the object is created recursively.

Although we developed the copy method only for bean-type and array-type objects, we can develop this deep copy method by reflection API for many types in advance. However, for the user-defined application-specific objects, it is difficult to develop deep copy method by using the reflection API.

C. Copying by the clone method

The clone method of the java.lang.Object class creates a new instance of the same type of object and copies all the reference values of its fields. If the application object implements the java.lang.Cloneable interface and its own clone method does not override the clone method of the Object class, the clone method of the application object implements a shallow copy. That is, any objects referred to by the fields are shared between the cloned object and the original object. In this case, if all of the fields of the application object are immutable types, the clone method is adequate to avoid side effects. If some mutable fields exist in the application object, the shallow copy is not

enough. If some mutable fields exist, a deep copy method which recursively copies the object tree is required.

In Web services, the interface of the service is described formally in a WSDL document. The data type of the message on the wire is defined by using a schema language like XML Schema in this WSDL document. The WSDL compiler in Apache-Axis generates Java classes for the data types that are defined by an XML Schema in WSDL. The generated classes are serializable and bean-type. Although the current WSDL compiler does not add clone methods, it should be easy for the WSDL compiler to add a proper deep clone method to generated classes. Therefore, when the client application uses the classes generated by a WSDL compiler as application objects, Java serialization, Copy by reflection and Copy by clone methods will be available.

4.2.4. Optimizations for read-only objects. Here, we consider when a deep copy is not required. This method does not copy the application object itself, but caches only the reference value of the application object. In this case, the reference value in the cache refers to the application object passed to the client application. When the application object modifies by the client application, then the cached object is also modified. However, there are application objects that do not need to be copied. Immutable objects, for example, String objects, primitive type values, and their wrapper objects cannot be changed. Therefore the cache can share these application objects with the client application.

Also, if the application object is read-only and is not modified by the client application, the cache can share the application object. In this case, the client application should explicitly assure the cache that the application object is read-only. It is hard for middleware to automatically detect that an application object is read-only in that client application. Therefore, the cache should be explicitly configured by an administrator of the client application when the application object is read-only.

5. Experiments and performance evaluation

In this section, we evaluated the performance of each of the methods that are described in the preceding section. First, we measured by using a micro-benchmark. Then we evaluated the performance in a possible scenario.

5.1. Micro-benchmark

In our micro-benchmark, we measured the processing time for cache key generation and cached data retrieval in cache hits. Cached data retrieval means the elapsed time for getting the application objects from the cached data.

The experimental environment was as follows. CPU: Pentium-4 1.80 GHz, Memory: 512 MB, OS: Windows XP Professional SP1, JVM: IBM JDK 1.4.1, Web services middleware: Apache-Axis 1.1, XML parser: Apache-Xerces 2.4.0.

The total time to perform 10,000 iterations for each method was measured. In these measurements, the benchmark made 10,000 accesses before measuring the time for another 10,000 accesses. This was done so that the compilation time of the JIT compiler would be excluded from the measured times. In Web services middleware, it is expected that the stable running state will continue for a long time. Therefore, it is more appropriate to measure the execution time after the JIT compiler has completely compiled the code.

In this experiment, the cached objects were in memory. We could store the XML messages and Java serialized forms on the hard disk, but disk access is slower than memory access. For fair comparison, we held all of the cached objects in memory.

For this experiment, we used the following three operations selected from Google Web services [19]. These operations have different types of return values, “small and simple”, “large and simple”, and “large and complex”, although the request parameters for these operations are not much different.

1. doSpellingSuggestion operation: This operation returns a single string object. That is to say, the returned object is very small and simple.
2. doGetCachedPage operation: This operation returns only a byte array containing a cached Web page from the Google server. The byte array is encoded using Base64 in a response XML message. This return value is relatively large but simple.
3. doGoogleSearch operation: The return value is a GoogleSearchResult application-specific object which encapsulates the complete results returned by the Google search. The GoogleSearchResult object has eleven fields. Nine fields are simple types, that is, String, int, double, or boolean. One field refers to the array of ResultElement objects and the last field refers to the array of DirectoryCategory objects. The ResultElement object has ten fields, nine simple types and one DirectoryCategory. The DirectoryCategory object has two String fields. Thus, this return value is large and very complex.

Table 5 summarizes these three operations with respect to request parameters and return values. Compared with the returned values, the request parameter types are not much different. Because a cacheable operation is a kind of retrieval operation, its parameter is likely to be simple. On the contrary, an operation that requires complex parame-

IEEE COMPUTER SOCIETY

ters may not be suitable for caching because cache hits are less likely to occur.

Table 5. Summary of the three Google operations

	Spelling Suggestion	Cached Page	Google Search
Request parameter objects	String x 2	String x 2	String x 6, int x 2, boolean x 2
Return value object	String (small and simple)	byte array (large and simple)	GoogleSearch Result (large and complex)

Tables 6 and 7 show the result of our micro-benchmark for cache key generation and cached data retrieval, respectively. The numbers in these tables are the times for single events. For this experiment, we modified the GoogleSearchResult objects so that all of the methods could be applied. By the modifications, these objects are serializable, have a deep clone method, and have a default constructor and public getter and setter methods for all fields.

Table 6. Processing times for cache key generation

(msec)	Spelling Suggestion	Cached Page	Google Search
XML message	0.213	0.212	0.298
Java serialization	0.021	0.022	0.036
toString method	0.005	0.005	0.008

Table 7. Processing times for cached data retrieval

(msec)	Spelling Suggestion	Cached Page	Google Search
XML message	0.299	0.708	3.244
SAX events sequence	0.094	0.458	1.986
Java serialization	0.014	0.046	0.276
Copy by reflection	n/a	0.019	0.046
Copy by clone	n/a	n/a	0.007
Pass by reference	0.001	0.001	0.001

From the experimental results for cache key generation, Java Serialization is ten times faster than the XML message approach. The toString method approach is extremely fast. As described above, a cacheable operation often has only simple type parameters. Therefore, the toString Method approach is optimal in many cases.

Before we discuss the performance of each approach for cached data retrieval, we describe the differences between the GoogleSearch and CachedPage operations. Although the sizes of the response XML messages for GoogleSearch and CachedPage are almost the same (see Table 9), overall the GoogleSearch is heavier than the CachedPage. A GoogleSearch has many elements in the XML message and a complex structured tree in the application object. On the other hand, a CachedPage has only one element in the XML message and only a single array of bytes in the application object. The test results show that a complex object requires more processing than a simple object, even if their sizes are the same.

From these experimental results, caching the SAX event sequences saves about half of the work compared to caching the XML messages. In Java serialization, the load for a cache hit is less than one-tenth of the load for caching an XML message. Copy by reflection is at least three times faster than Java serialization. Copy by clone is very fast compared with reflection, in spite of providing essentially the same function. Pass by reference has little overhead.

The methods evaluated in this experiment are arranged sequentially from highest to lowest performance as follows: 1) Pass by reference, 2) Copy by clone, 3) Copy by reflection, 4) Java serialization, 5) SAX event sequence, and 6) XML message. Because these methods have different limitations, each method should be used according to the type of the application objects.

In addition, we measured the memory sizes of the cache keys and cached objects for each operation. Tables 8 and 9 show these results.

Table 8. Memory size of cache keys

(bytes)	Spelling Suggestion	Cached Page	Google Search
XML message	586	579	974
Java serialized form	234	238	462
Concatenated string	120	123	164

Table 9. Memory size of cached objects

(bytes)	Spelling Suggestion	Cached Page	Google Search
XML message	520	5,338	5,024
Java serialized form	21	3,611	1,914
Java object	28	3,600	464

In this experiment, the cached objects were stored in memory. The Java serialization form and the Java object were much smaller than the XML message, except for the CachedPage operation. Since the CachedPage includes only a single array of bytes, the size of the object is not very different for the different data representations. Although we did not experiment with any capacity stress testing for the cache memory area, it is desirable that the memory usage be small.

5.2. Evaluation by portal site scenario

In addition, we evaluated each cache method for a portal site scenario. In Figure 2, the configuration of the scenario is shown. We developed dummy Google Web services for the test. We also developed a Web portal site

IEEE COMPUTER SOCIETY

that uses the dummy Google services as its back-end services. The Web portal site works with the Web services client middleware (Apache-Axis) with caching. Then we stressed the Web portal site using a load simulator.

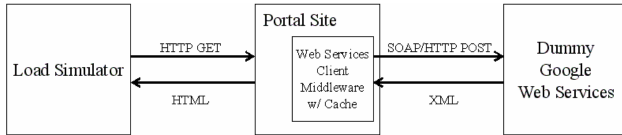

Figure 2. Portal site scenario configuration

We used Jakarta-Tomcat 4.1.24 as the servlet engine for the portal site and for the dummy Google Web services. We used Web Performance Tool 1.9.4.1 at IBM alphaWorks as the load simulator. The environment for Web services middleware is the same as used for the micro-benchmark above.

The dummy Google Web services actually return the same response XML messages every time in order to insure that the dummy services are not too demanding. This is because we wanted to evaluate the performance of the Web services client cache on the portal site, so the back-end services should not be a performance bottleneck. If back-end services were a performance bottleneck, then the performance of the portal site would depend mostly on the cache-hit ratio.

We evaluated the GoogleSearch operation because the difference of performance there among each method is larger than for the other operations. We used the toString method approach for cache key generation. We then compared each cache approach for cached data retrieval and artificially changed the cache-hit ratio from 0% to 100%.

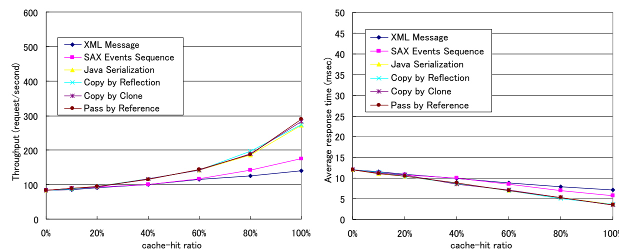

Figure 3. Throughput and average response time without concurrent access

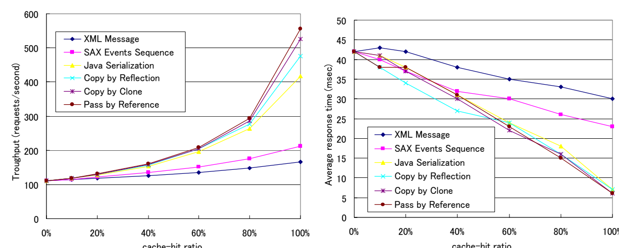

Figure 4. Throughput and average response time with 25 concurrent accesses

Figures 3 and 4 show the throughput and average response times to the portal site from the load simulator. First, we stressed the portal site without concurrent access. In other words, the next request was not issued until after the reply was received from the current request. In this experiment, the CPU utilization rate of the portal site machine was 50-70%. The results of this experiment are shown in Figure 3. In these results, the throughput with a 100% cache-hit ratio while caching XML messages, SAX events sequences, and application objects become approximately 1.5 times, 2 times and 3 times larger than with a 0% cache-hit ratio, respectively. The response times were 1.5 times, 2 times and 3 times shorter, respectively. The four methods of caching application objects were not very different from each other. Because the absolute value of the processing time is relatively small, the differences seem to be hidden by other processes.

Next, we stressed the portal site with concurrent accesses. At most 25 requests were issued simultaneously. In this experiment, the CPU utilization rate of the portal site machine was over 95%. The results of this experiment are shown in Figure 4. In these results, overall, the throughput was larger and response times were longer than in Figure 3. In addition, it is important that the throughput with a 100% cache-hit ratio while caching application objects was 5 times larger than with a 0% cache-hit ratio and that the response time was 8 times shorter. This indicates that caching application objects is very effective, especially in a heavy traffic situation.

6. Optimal configuration

In this paper, we described several methods for the client cache. These methods have large differences in their performance, and various limitations. It is important to combine these methods properly in order to develop a high performance cache. In this section, we describe a configuration in which the client application does not need to be at all conscious of how the response data is cached, and does not have any limitations on its application objects.

The method using XML messages is the simplest mechanism among the methods evaluated in this paper. In this method, there are no limitations on the application objects. However, the performance of this method is not good. The improved version using SAX events sequences also does not have any limitations on the target application objects.

Although the Java serialization method has the limitation that the target application objects tree should be fully serializable, middleware can automatically detect whether or not the application object is serializable. Also, if the target application objects are bean types, array types, or

IEEE COMPUTER SOCIETY

the types that we developed in advance their copy methods with the reflection API, then we can use the copy by reflection method. In addition, immutable objects whose types are known in advance to be immutable, for example String, can be shared between the cache and the client application without any change.

We summarize the optimal cache methods as follows:

a) Immutable types like String and primitive types:
   - Pass by reference,

b) Bean-type objects and array-type objects:
   - Copy by reflection,

c) Serializable objects:
   - Java serialization,

d) Types not included above in a), b), or c):
   - SAX event sequences.

At run time, middleware can dynamically classify the target objects within the above list without requiring any configuration by an administrator of client applications.

7. Concluding remarks

In this paper, we discussed the design of efficient response cache mechanisms in the Web services client middleware. Although caching application objects will improve the cache performance, the cache must avoid side effects causes by the changes of the application data by the client application. We described the caching using XML messages and improvements by caching the SAX event sequences. We also proposed caching application objects using Java serialization, reflection copy, and clone copy. Then we showed an optimization for read-only objects. We measured the performance of each method for cache hits. Since these methods have large performance differences and various limitations, we suggest a configuration to combine these methods properly in order to develop a high performance client cache.

Acknowledgements

We would like to thank Yuichi Nakamura, Yasumasa Kajinaga, and many other members of the IBM Tokyo Research Laboratory for many valuable discussions and helpful comments. We also thank Shannon Jacobs for his great efforts to fix numerous English problems in this paper.

References

[1] C. Liu, and P. Cao. “Maintaining Strong Cache Consistency in the World-Wide Web”. In *Proc. IEEE 17th Int. Conf. on Distributed Computing Systems*, pp. 12-21, May 1997.

[2] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. “Engineering Server-Driven Consistency for Large Scale Dynamic Web Services”. In *Proc. WWW10*, Hong Kong, May 1-5, 2001.

[3] H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Ruhl, and F. Kaashoek. “Performance Evaluation of the Orca Shared Object System”. *ACM Transactions on Computer Systems*, Vol. 16, No. 1, pp. 1-40, 1998.

[4] C. Kohlhoff, and R. Steele. “Evaluating SOAP for High Performance Business Applications: Real-Time Trading Systems”. In *Proc. of the 12th Int. WWW Conf.*, May 2003.

[5] P. Cao, and S. Irani. “Cost-Aware WWW Proxy Caching Algorithms”. In *Proc. USEITS '97* (USENIX Symposium on Internet Technologies and Systems), December, 1997.

[6] J. Challenger, A. Iyengar, and P. Dantzig. “A Scalable System for Consistently Caching Dynamic Web Data”. In *Proc. INFOCOM '99*, March, 1999.

[7] A. Chankhunthod, P.B. Danzig, C. Neerdaels, M.F. Schwartz, and K.J. Worrel. “A Hierarchical Internet Object Cache”. In *Proc. USENIX '96*, January 1996.

[8] A. Delis and N. Roussopoulos. “Performance and Scalability of Client-Server Database Architectures”. In *Proc. 18th Int. Conf. on Very Large Databases*, 1992.

[9] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. “Adaptive Push-Pull: Disseminating Dynamic Web Data”. In *Proc. of the 10th Int. WWW Conf.*, pp. 265-274, Hong Kong, China, May 2001.

[10] L. Degenaro, A. Iyengar, I. Lipkind, and I. Rouvellou. “A Middleware System Which Intelligently Caches Query Results”. In *Proc. of Middleware 2000*, pp. 24-44, April 2000.

[11] J. Eberhard, and A. Tripathi. “Efficient Object Caching for Distributed Java RMI Applications”. In *Proc. Middleware 2001*, LNCS 2218, pp. 15-35, 2001.

[12] R. Tewari, M. Dahlin, H. Vin, and J. Kay. “Design considerations for distributed caching on the Internet”. In *Proc. IEEE 19th Int. Conf. on Distributed Computing Systems*, pp. 273-284, 1999.

[13] C. Gray, and D. Cheriton. “Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency”. In *Proc. SOSP '89* (12th ACM Symposium on Operating Systems Principles), pp. 202-210, 1989.

[14] A. Iyengar, and J. Challenger. “Improving Web server performance by caching dynamic data”. In *Proc. USENIX Symp. on Internet Technologies and Systems*, December 1997, pp. 49-60.

[15] G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. “Implementing a Caching Service for Distributed CORBA Objects”. In *Proc. of Middleware 2000*, pp. 1-23, April 2000.

[16] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. “Extensible Markup Language (XML) 1.0 (Second Edition)”. W3C Recommendation, October 2000, http://www.w3.org/TR/REC-xml

[17] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielson, S. Thatte, and D. Winer. “Simple Object Access Protocol (SOAP) 1.1”. W3C Note, May 2000, http://www.w3.org/TR/SOAP/

[18] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. “Web Services Description Language (WSDL) 1.1”. W3C Note, March 2001, http://www.w3.org/TR/wsdl

[19] Google Web APIs (beta), http://www.google.com/apis/

[20] Amazon Web Services, http://www.amazon.com/webservices/

[21] The Apache Software Foundation, Apache Axis, http://ws.apache.org/axis/

1063-6927/04 $20.00 © 2004 **IEEE**

IEEE COMPUTER SOCIETY