Optimizing web services performance using cache

Julio Fernandez Vilas, Ana Fernandez Vilas, Jose Pazos Arias

Campus Universitario s/n, 36200 Vigo, Spain
E-mail: {jfvilas,Ana.Vilas,jose}@det.uvigo.es

[Received 00/00/00; accepted 00/00/00]

Abstract. Due to the complex infrastructure of web applications (using application servers, web servers, database servers, etc.) response times perceived by clients can be significantly larger than they would like. The same can be applied to web services that have been implemented using the same web technology. On the other side, repeated invocations to the same "read-only" web service with the same request data may often produce the same response from the provider, what can be considered a useless work. In the search for a performance improvement, we propose a cache-based extension to the architecture that can enhance current web services architecture.

Keywords: Web Services, Performance, Caching, Consistency, Virtualization.

1. Introduction

It is easy to understand how to implement a caching system when we talk about web pages (static information). However, where (and how) can we apply caching techniques to web services entirely based on the use of dynamic information? We have developed a solution to this problem, but it is important to take into account the fact that caching techniques can be applied to web services only under specific circumstances. For a web service to be considered a cacheable web service, following facts should be true about it:

- The number of invocation requests is large enough. If so, there will be a significant increase in performance when using a caching system.
- There is at least a method (a WSDL port, let us call it a "cacheable method") whose response to a request can be repeated along time without any variation in SOAP responses. If not, it may not be adequate for implementing caching mechanisms.
- The results received by clients invoking a cacheable method when the response is cached, must not differ from those received by clients when the response is not cached.

Different applications in real life can be eligible to be implemented using cacheable web services. Some examples of cacheable web services are:

- A web service that provides a weather forecast, where changes in responses are not expected to occur in a short time.
- A web service to obtain stock market share prices. Although this is a typical real time application (clients obtain real prices on each invocation), consider how the service runs when there are no changes to price, or when the stock market is close.
- A web service to obtain the status of an item in relation to a stock management service (available quantity, type, model, etc.).

After this introduction, the rest of the paper is structured as follows: in section 2 we analyze our proposed architecture for caching, while section 3 deals with its implementation. Sections 4 and 5 deal with caching methods and its message patterns. Section 6 holds a specific analysis on 3LCA. Section 6 exposes our experiments in implementing caching methods.

2. Architecture for caching

It is possible to define at least to different caching structures in web services architecture. The first one is a two level caching architecture (2LCA), which involves the existence of two entities: server and client. This caching architecture is not new, and caching is applied here using a typical caching implementation, like the one used by web browsers that access web servers directly, for instance.

Three-level caching architecture (3LCA) is also possible. This kind of architecture is likely to occur in an environment where intermediate elements are part of the whole architecture. In a 3LCA, a third component appears, the intermediate element, who can also be an active actor in the caching system. A sample implementation of this caching architecture is the one used in web proxies, which are the intermediate elements between web browsers and web servers.

We can differentiate two main types of 3LCA depending on the implementation of each element in the architecture:

- Three-level homogeneous caching architecture (HoCA): all three entities in the architecture communicate using the same protocols. For example, a web client, a web proxy and a web server is a good
example of HoCA. In web services architecture, a web service, a client application, and an intermediate web node, like a SOAP router or a VWS engine [1] would form another good example.

- Three-level heterogeneous caching architecture (HeCA): two entities communicate using one communication protocol, while the third one uses a different protocol. For example, consider a fat client invoking a remote component (using DCOM or EJB, for example). Let us suppose that the remote component invokes a web service (the third entity). The remote component and the web service communicate using a protocol different to the one used between the fat client and the remote component.

It is important to make a distinction between HoCA and HeCA, because the type of architecture will determine the way a caching technique can be applied. In a HeCA, only one level of web service caching can be applied. Of course, ad-hoc caching mechanisms can be implemented, but may not adhere to standards, and they are implementation-dependent. On the other side, two levels of caching can be implemented in HoCA: a first level cache between providers and intermediate elements, and a second level cache between intermediate elements and clients. The main difference between HeCA and HoCA from the caching-architectures point of view is that HoCa allows the propagation of SOAP messages without a problem, so client applications and intermediary elements keep the same caching information about cached messages.

n-level caching architecture is also possible, but it is always a combination of two-level and/or three-level systems. n-level caching architecture can also be viewed as a hierarchical caching architecture, where clients are the leaves, the web service is the root, and intermediate elements such as proxies, SOAP routers or VWS engines, are the intermediate nodes of the hierarchical tree. Hierarchical-cache-based solutions have already been proposed in [2] and [3] for static web content.

3. Implementation

The base for the implementation of our proposed caching architecture is the use of SOAP-Header elements. Like other standards, we propose to add specific caching information to the header of SOAP messages. The information contained in a SOAP header must obey a language in order to use our cache purpose: VWSEL (Virtual Web Services Extension Language), created as an extension of our VWSIDL (Virtual Web Service Definition Language) [1].

It is responsibility of every web service to decide what information will be included inside the VWSEL section of SOAP messages. Web services should use VWSEL documents to send caching information to clients. A VWSEL document can be used to express an interval where the SOAP message carrying that document is valid. Therefore, a subsequent invocation of the same service with the same invocation data in the abovementioned period would be useless, since the response will not vary.

To implement our caching architecture, it is needed that the client entity of an invocation (a client application or an intermediary node) understands VWSEL. If a client application knows nothing about VWSEL, the web service will not send any VWSEL information with SOAP responses when invoked from that client. However, if a client understands VWSEL, it should send a VWSEL document notifying the web service that situation, so the web service can take advantage of the use of caching mechanisms.

Although caching is not new, we detect a need for a caching mechanism in the current web services architecture, since web services should not depend on its transport layer to use caching mechanisms. The responsible of deciding what-to and what-not-to cache should not be the underlying protocol, as proposed in [4], but the web service itself. What would occur to HTTP-cache-based performance optimization if the transport layer stops being HTTP-based to be anything-else-based?

There is a very important difference between using HTTP caching for web services [5] and our purpose. How do underlying protocols differentiate between responses originated at different ports? Moreover, how can different invocations to the same port of a web-service produce different caching-information response? What we propose is to give web service developers the opportunity to control the caching process from web-services source code. This way, different invocations to same port of a web service can have different caching responses. Let us think about a web service querying stock market share prices. When the market is closed, the response of the port can be cached without a problem until market opens again. When the market is open, caching should be reduced to a minimum or even not used.

4. Caching methods

Using caching effectively requires updating cached messages, in order to maintain consistency. Two types of consistency can be defined [6]: weak and strong. In weak consistency, a stale SOAP response can be returned to a client application in response to a cached invocation. In strong consistency, a stale copy of a cached response will never be sent to a client application.

Two consistency models require the use of two caching methods: weak caching and strong caching. In weak caching the web services must send "expire" information to clients, so clients can use that information to decide whether to use cached responses or invoke the web service. "Expire" information is based on the use of a Time-To-Live (TTL) mechanism.

Implementing weak caching is easy for web services and for web clients, but it can produce the use of incorrect SOAP responses in applications. For web services, unlike traditional HTTP-based applications, is vital to maintain consistency, so a strong caching method should be used if
There are only two basic mechanisms that allow the creation of a strong caching method: invalidation and polling. Using polling, client applications are responsible of querying web services about the freshness of SOAP response messages. In order to implement a strong consistency model using polling, client applications are required to perform a poll to the web service every time an invocation is to be started. Please note that every invocation produces a SOAP message sent from the client entity to the web service, so network latency is always present. However, this could be a useful mechanism when considering long SOAP response messages, due to the network optimization obtained by using caching. It could also be useful when the web services processing-time is too long.

The second mechanism, invalidation ([7], [8], [9], [10]), is a more complex strong cache method, where web service notifies clients about the validity of a concrete SOAP message. Clients do not have to care about the validity until they receive a notification invalidating the SOAP message. To be able to implement this mechanism, servers (where web services run) must have access (directly or indirectly) to clients, in order to be able to send notifications.

However, invalidation can produce invalid responses, due, for example, to the impossibility of the web service to notify clients (a network outage, for instance) about the existence of an invalid SOAP message.

Our VWSEL purpose supports two caching methods. First, a simple weak caching is supported in order to deal with two- and three-level architectures, where updates are not likely to occur frequently. The mechanism for implementing weak caching is the abovementioned TTL.

Second, a strong caching method is provided by extending VWSEL with WS-Addressing [11]. Evidently, using WS-Addressing inside VWSEL is just a way to make specifications, that is, web services need to have access to client entities (client applications or intermediate elements). Due to this need, strong caching is easy to implement in HeCA and HoCA, since the use of intermediate elements (with similar functions to proxy servers) simplifies the network implementation.

When using 3LCA, it can be a good practice to mix the two caching methods. Use a strong caching method as a first-level cache (between web service and intermediate element), and a weak caching method as a second level cache (between an intermediate element, such as a SOAP router or a VWS engine, and client applications).

How do the web services invalidate cached SOAP messages? We propose the use of two cache-invalidation mechanisms. The first, relaxed invalidation, is based on the use of notifications that web services send to clients when a SOAP response has been changed. Web service is not expected to receive any answer from clients. It is responsibility of clients to delete cached message. It is clear that this mechanism is valid only under certain circumstances, when using "almost-read-only" web services.

The second invalidation mechanism, secure invalidation, is based on the concurrent use of two methods to invalidate messages: a Time-To-Live and a notification with response. This is what we call iTTL (invalidation with TTL). When a web service wants to invalidate a SOAP message that has been previously cached by clients, it first sends a notification to all of them, and waits for their responses. If one or more clients do not respond to notification, the web service must wait for the TTL to expire. Obviously, cached messages on client entities have also a TTL (sent by the web service with each message). The message TTL must be less or equal to the notification TTL used by web service. When the web service TTL has expired, TTL on all cached copies of the message have also expired, and thus, a new copy of the SOAP message can be sent to client entities as requested.

A similar algorithm has been proposed in [12], but adding a heartbeat to control the status of the communication between client caches and servers, what is very likely to increase bandwidth utilization.

5. Message patterns

Client applications must understand VWSEL in order to use our caching purpose, but this is not a disruptive restriction. When a client application performs an invocation, it must attach a VWSEL document specifying what type of caching it supports. On the other side, a web service must extract VWSEL information in order to produce the appropriate caching information and attach it to the SOAP response. Please note that if a client is not VWSEL-capable, the invocation will work without a problem, since the web service will not send any caching information to client. On the other side, if a web service is not VWSEL-capable, it will not extract any VWSEL information and it will send the SOAP response as usual.

As a sample, figure 1 shows a SOAP request with caching information requesting a secure invalidation caching method. WS-Addressing is used to specify the URL where invalidations will be received. Figure 5 shows the response to the previous invocation specifying a caching period of 10 minutes. And figure 6 shows an invalidation message sent to clients that have cached the previous SOAP message.

6. Optimizing 3LCA

3LCA is a good opportunity for an easy caching mechanism implementation. As stated before, when using HeCA, different entities in the caching architecture use...
different communications protocols. From this point of view, we can split a HeCA into two 2LCA, one of them using standard protocols like SOAP, and the other one using a caching mechanism specifically designed for a non-standard communication protocol.

On the other hand, HoCA can also be divided into a pair of 2LCA using standard protocols. This way, caching methods like those explained in previous section can be applied. But the fact that HoCA uses the same communication protocol on all three entities in the caching architecture is a great opportunity, since solutions different to those of 2LCA can be specifically applied to 3LCA.

As part of a bigger work, we have developed a new architecture for web services, fully compatible with current one. Our purpose is based on the use of a technique already in use in other disciplines like VLAN [13] or hardware enhancements [14]: virtualization.

We have implemented and described solutions to other common problems in web services technology like high availability [1] and Quality of Service [15] using our virtualization-based purpose.

Virtualization is based on grouping one or more web services inside a unique wrapper, which is then published as a standard web service. Clients use the new virtual web service as a standard one, i.e. there is no difference between real and virtual from the clients point of view. With virtualization, some additional logic can be performed out of the client applications (error management, provider selection, caching, etc.), and this way the client’s software complexity is reduced, while performance enhancements can be achieved.

The wrapper term is used here to refer to a virtual view of a collection of web services, so clients have a unique view of the collection. The virtual view is in fact published as a standard WSDL document. Virtualizing a web service requires a change in the standard web services architecture, since a virtual web service must reside in an intermediate element different from the client and the provider.

In the standard web services architecture, clients use “direct invocations” to invoke web services. What we propose is to introduce a change in the architecture, moving the invocation model from direct to indirect.

The architecture we propose is mainly based on the introduction of a virtualization layer. This layer must be understood as non-intrusive element; in other words, its introduction must respect, at least, the following directives:

- It must not alter the current infrastructure (protocols, languages...).
- It must not affect the web services over which it is built. There must be no need to modify a web service to adapt it to the new architecture.
- It must not affect client applications.
- It must not alter the current operation of the providers, allowing a web service to comply with the standard architecture and with the new one at the same time.

The use of the virtualization layer together with the indirect invocation model materialize in what we have described here as a HoCA.

6.1. Components in virtualization

The architecture we propose for the use of virtual web services determines the existence of five entities:

1. **Client**: the entity that needs a service. This is equivalent to a client in a SOA structure [16].

2. **Delegate client**: the agent on whom the client delegates the responsibility for executing a service. Generally speaking, it represents client applications.

3. **Provider or service provider**: the entity that offers a service. It must be seen as a provider in a SOA architecture.

4. **Delegate provider**: the agent on whom the provider delegates the responsibility for offering a service. In web services technology, a delegate provider is a web service.
5 **Engine or intermediary**: the entity in charge of putting delegate clients and delegate providers in contact. It performs communication processes between both delegates following different algorithms, and pursuing different objectives depending on which is the runtime environment.

The VWS engine can be something as simple as adding some kind of decision capabilities to the client proxies, but it can be something as complex as a dedicated server (figure 4).

The VWS engine is not a virtual server. It is a standard one, and it should be implemented the same way as a server used to process standard web service invocations. The term "VWS engine" refers to the capability of a server to understand virtual services definitions. That is, it can receive, process, and respond to standard web service invocations. In order to process a request, the engine uses VWS descriptions to select and invoke the most suitable web service provider.

7. Experiences with caching

We have implemented a three-level HoCA. We have built a prototype that implements our VWS purpose. We have also built a test case with the structure shown in figure 5. On the server side, we have an IBM mainframe (a z900-303, with 1300 MIPS). We use a SOAP front-end processor to convert SOAP messages into plain text that can be redirected to the mainframe, to a CICS region [17]. In the middle tier, we have installed our VWS engine in a Wintel machine. On the client side, we have developed and installed a .Net application responsible of performing a massive web-service invocation process. The LAN connecting all these machines is a 100Mb Ethernet.

The software configuration is as follows. We have selected a financial CICS transaction on the mainframe consisting on a query on the daily net asset value (NAV) of about 300 different mutual funds. This transaction has been converted into a web service by using a SOAP preprocessor, which transforms SOAP messages into plain text and vice versa. The client application can invoke the web service requesting the NAV of different funds at different dates. The VWS engine consists on our VWS engine prototype with a virtual web service build around the abovementioned web service in mainframe.

The test case consists on successive executions of client application, each one consisting on 20,000 continuous invocations to the virtual web service. The implementation web service performs always the same kind of query with different data. The specification of the different tests is explained next:

- Test 1: Query on the same fund on different dates (within the same month).
- Test 2: Query on different funds on the same date.
- Test 3: Query on different funds on different dates (within the same month).

The workload created by our “massive-invocation” client application is similar to that of a lot of branch offices belonging to a bank or an insurance company. All three tests were run twice with two different engine configurations: with and without caching. The caching mechanism used is a weak caching with a 60 seconds TTL.

The results of the tests are summarized in figure 6. As we can see, response times perceived by clients are very different whether we are using cache or not. When
cache is not in use (figs. 7, 8, 9), the average response time (white line), typically ranges around 260 milliseconds. When using cache (figs. 10, 11, 12), the average response time is about 30 milliseconds, once responses have been cached (after an initial "fill-the-cache" period). Even when TTL of cached messages have expired, and the cache starts to be refilled, the average response time keeps very low (when compared to "no-cache" configuration).

8. Related work

There are some works on improving web services performance by using caching; some of them are based on several different programming mechanisms. In [18], we can see a solution based on the use of a cache object. This object is a Java object that must be managed from the web service logic. This solution does not avoid network traffic, since invocation must be received at application code, in order to access the abovementioned cache object.

Other kinds of solutions are based on the use of HTTP headers to manage the expiration of the HTTP responses. In [4], it is proposed the use of programming attributes (similar to compilation directives). The use of the WebMethod attribute, together with the CacheDuration property, allows a simple way to control the TTL of the response. Again, this solution is based on the use of HTTP, enforcing a dependency between a web service (and its business logic) and a transport layer (HTTP). In addition, it is a static mechanism, that is, programmers cannot control the duration of a response depending on the logic of the web service.

The work we have shown here, has a close relation with the theories exposed in [6] and [19]. Moreover, the same as the theories exposed in [2] and [3], our caching purpose will add even greater benefits when applied to a hierarchic architecture, like the one we proposed in [1], based on the use of VWS engines in charge of routing SOAP messages.

Our work proposes a different solution to implement caching than the one exposed in [20]. The main difference take root in the level of caching used: while [20] is targeted to WSDL ports, we propose to cache SOAP messages, and only the SOAP-Body part of messages, ignoring transport headers. Additionally, in [20], the consistency model is based on a request-response mechanism where caching entities must query the web service for the validity of a concrete SOAP message. This may be a good solution for read-only web services.

Some of the ideas used to build our architecture are closely related to other ideas extracted from different works on different disciplines, like Content Delivery Network [21]. CDN technology experiences problems similar to web services technology when taking into account consistency in cached environments. We have also worked on invalidation looking at works that study consistency in distributed file systems [8], and disconnected mobile environments ([9], [10]).
9. Conclusions and future work

Our purpose is a very good solution to improve the performance of web-service-based applications, especially in stressed environments (with high volume of invocations). In addition, our purpose, far from the HTTP cache system, gives a high degree of freedom to web service developers, since our purpose, based on the use of extra information that is sent with SOAP messages, allows web service developers to control the way that messages are cached by client entities. That is, our purpose allows developers to control cache system in a different way on each invocation, depending on the execution logic of the web service.

In addition, our purpose is not limited to the use of a TTL system, like the one used by HTTP, our purpose goes further, and it allows the creation of a dialogue (that must be previously negotiated) between client applications and web services. This negotiation allows reducing the number of invocations to a minimum while maintaining a strong consistency. It is important to note that there is not a unique mechanism for controlling cache. The negotiation between client and web service allows the use of TTL, polling, invalidation, etc.

The experiments we have performed within our test case show a successful implementation of our proposed caching system. The results show clearly that, when applied to some types of applications, a simple cache system is a good way to improve the performance of the whole system: low resource consumption on web server executing web services, better response time to client applications, and low network bandwidth utilization. That is, the caching system is not only good for web service, but also for the rest of components in web services architecture.

It is important to note that the “cacheable” concept is applied to ports, and not to web services. So the number of web services where caching can be applied is potentially very large, since caching is used at port level, not at service level.

We wish to note that the performance of a strong consistency method, like secure invalidation, is highly dependent on the way the iTTL mechanism is implemented. The values for TTL will impact greatly on the performance of the whole system: a good starting point is to use Adaptive TTL [22] together with invalidation instead of simple TTL, or any other mechanism that can help in predicting the lifetime of a cached SOAP message.

It has not been discussed here, and should be part of a future work, but the algorithms used to manage a web cache (LRU, LFU,...) should be analyzed and adapted to the new environment of web services. A similar analysis should be undertaken in relation with cache refreshment policies [23], thus creating active client and/or intermediary caches when applicable. Additionally, as part of a future work the privacy of SOAP messages should be taken into account when using intermediary cache structures.

Other invalidation schemas could be adequate. An invalidation channel as proposed in [24] could be used, but possibly increasing bandwidth utilization.

References:


Name:
Julio Fernandez Vilas

Affiliation:
Telematic Engineering Department, University of Vigo

Address:
Campus Universitario s/n, 36200 Vigo, Spain

Brief Biographical History:
Working out on a Ph.D since 2002. Director of Mainframe System Administration Department, Galician Savings Bank (Caixa Galicia)