Dynamic Invocation of Replicated Web Services

José Airton Fernandes da Silva, Nabor das Chagas Mendonça
Mestrado em Informática Aplicada, Universidade de Fortaleza (UNIFOR)
Av. Washington Soares, 1321, Fortaleza – CE, Brasil
{jairton.mia, nabor}@unifor.br

Abstract

This paper presents a framework for dynamic invocation of geographically replicated web services. The framework implements a transparent invocation process, based on different server selection policies, which aims at reducing service response time at the client side. Preliminary experimental results, obtained in real-world Internet setting, show that, in addition to the individual capacity of each server, the service invocation process is affected mainly by client characteristics such as connection bandwidth and workload distribution throughout the day.

1. Introduction

Web Services [6] are a standardization of the emerging Service Oriented Computing paradigm [14], and constitute a powerful mechanism for integrating distributed applications over the Web, independently of programming language, execution environment, and communication protocol. More than simple web resources (e.g., HTML files, images and GCI scripts), web services are loosely coupled applications that can be located and accessed using standard XML-based technologies, such as SOAP [16] (a message exchange protocol), WSDL [19] (a language for describing service interfaces), and UDDI [18] (a repository for dynamic service discovery and location).

Web services technologies are being deployed within a scenario where providing quality of service guarantees is fundamental to the success of most organizations, especially given the ever increasing demand for web resources [7]. In this scenario, replication can play an important role. It may be used to increase system availability, by providing resources redundantly at multiple sites; reduce access time, by accessing a nearby copy; and also increase system throughput, by distributing client requests amongst multiple servers [5].

Two main approaches have been traditionally used to control access to replicated resources on the Internet. The first approach includes mechanisms for automatic distribution of client requests at the server side. Since its primary focus is on reducing access time through load balancing, this approach is mostly used in cases where servers holding replicas for the same logical resource are tightly coupled (either physically or administratively) to each other, such as in service clusters [8]. The second approach involves mechanism that are incorporated into the client applications themselves, or to nearby proxy servers, and which aim at exploring more effective ways of accessing a replicated resource taking into account the specific characteristics of each client (e.g., [20][1][10][9][15]). Hence, this approach is more commonly used in cases where replicas are geographically apart from each other.

In our work, we focus on the second approach. In particular, we are interested in investigating client-side mechanisms for locating and improving response time of geographically replicated web services. It should be noted that service replication also involves other important issues, such as replica allocation and synchronization. Since those are typically implemented at the server side, they are not considered in this paper.

When accessing a replicated web resource, a crucial issue is selecting the server with the best replica, i.e., the replica that will potentially provide the fastest response time to the client application. This is not a trivial choice, though. Variations in server workloads, network latencies and client connection characteristics, amongst others, are decisive factors that can affect the server selection process for a given client application in a given instant. In view of those difficulties, several works have proposed and evaluated different server selection policies (e.g., [1][10][9][15]) for accessing replicated web resources at the client-side. However, most of the proposed policies were implemented only considering web resources in the form of HTML files and images. Although conceptually applicable to the web services domain, in practice those policies do not allow their direct adaptation to this new context – at least not the way they have been described – due to the intrinsic characteristics of web services, which are closer to the execution model of distributed objects than to the retrieval of documents and images from a remote web server.
More recently, a few works have started to address the issue of applying server selection strategies to dynamically invoke functionally equivalent web services [11][3][13]. However, neither of those works has offered any empirical evidence of the benefits of their approaches in practice, nor have they compared their proposed server selection strategies with those already described for accessing traditional web resources.

To address the above issues, in this paper we present the Replicated Web Service (RWS) framework for dynamic invocation of geographically replicated web services. In contrast to most web service frameworks currently in use (such as Apache’s AXIS [2], Microsoft’s .NET [12] and SUN’s JWSDP [17]), the RWS framework implements a transparent invocation process, based on different server selection policies, which aims at reducing service response time at the client side. The framework was used to implement and empirically evaluated five server selection policies. The evaluation was carried out using a “real world” replicated web service with four replicas distributed over three continents. Overall, the results show that, in addition to the individual capacity of each server, the replica selection process is affected mainly by client characteristics such as connection bandwidth and workload distribution throughout the day.

The rest of the paper is organized as follows. In the next section, we describe the RWS framework in more details. In section 3, we report on the method and results of the empirical evaluation. In section 4, we cover related work. Finally, we conclude the paper in section 5 with a summary of our research and some directions for future work.

2. The RWS framework

In this section, we present the main features of the RWS framework, including the replica location process, some selected implementation details, and the five server selection policies investigated.

2.1. The replica location process

At the client side, selecting the best replica of a replicated web resource requires, amongst other information, identifying the location of each server holding a replica of the resource. In the case of a replicated web service, replica location information can be obtained directly from the service’s WSDL description [19].

In a WSDL document, all information necessary for locating and invoking a given service is described in the <service> element. This element includes a <port> sub-element describing the binding protocol (HTTP, SMTP, etc) through which the service can be accessed. The <port> element in turn includes an <address> sub-element with the service access URL as one of its attributes. Since a given service may be available via different binding protocols, the WSDL specification allows the definition of multiple <port> elements for the same service. On the other hand, the specification imposes no restriction on the value of the <address> element of each <port>. Hence, by adding multiple <port> elements with different address values into the same <service> element, it is possible to define multiple access URLs for a given (replicated) web service, with each URL representing the location of a different replica. Figure 1 shows an example of a WSDL fragment describing a replicated web service with two replicas, with the access URL for each replica being described using a different <port> element.

In our work we assume that replica location information is represented in the replicated service’s own WSDL description, as explained above.

2.2. Selected implementation details

Once all replicas of a replicated web service have been identified, from a strictly performance perspective what matters for a client application is that each service invocation be responded as quickly as possible, regardless of which replica is actually being invoked. To make the invocation process transparent for the application programmer, an automatic replica selection mechanism has to be implemented and incorporated into the application’s underlying infrastructure. This could be done in two ways: either by extending the service invocation layer (or stub) that is incorporated into the client application by the underlying web service framework, or in the form of a proxy service that sits between the client application and the remote replicas. Our work on the RWS framework follows the
former approach, as it imposes a lower overhead on the client application and is also easier to deploy.

Specifically, the RWS framework was implemented by extending the client-side infrastructure of AXIS [2], a web service framework for Java. Our choice for AXIS was due to the platform-independent nature of Java, and also the fact that it is a well-documented open source project. Our extension involved the modification of the AXIS stub generation process, so that the generated classes could accommodate the replica location information available in the replicated service’s WSDL description. We also introduced two new components into the original AXIS infrastructure, namely the Selection Manager and the Invocation Manager, as shown in Figure 2. Those two components implement the mechanisms responsible for server selection and service invocation, respectively.

With the above modifications, the RWS framework can offer client application programmers, in addition to the native client side AXIS resources, the option of either explicitly specifying the service replica to be invoked, or delegating that decision to the framework itself, based on a given server selection policy. In the latter case, the framework selects the “best” replica dynamically (according to the criteria defined by the policy), every time a new service invocation is requested by the application. The client application, the set of generated classes, and the new components introduced by the RWS framework together constitute a smart client infrastructure similar to the one proposed in [20].

2.3. Server selection policies

In its current version, the RWS framework offers support for five server selection policies, namely Random Selection, Parallel Invocation, HTT Ping (or Probe), Best Last, and Best Median. They are described in more details below.

Random Selection – In this policy, the server to be invoked is selected randomly amongst the set of servers hosting replicas for the replicated web service.

Parallel Invocation – In this policy, all replicas are invoked “in parallel” (in fact, concurrently). The first response to be received in full is returned to the client application, with all pending invocations being interrupted and their partial response, if available, being discarded.

HTT Ping (or Probe) – In this policy, a “probe” (in fact, a HTTP HEAD message whose typical response size is in the order of hundreds of bytes) is initially sent to all replicas concurrently. The first server to respond the probe is then selected for invocation, with all later responses being ignored. This policy is a variation on the TCPing policy, proposed in [9], which uses a TCP-based probe mechanism for accessing traditional web resources.

Best Last – In this policy the server to be invoked is selected based on the performance of the last successful invocation of each service replica. Information on past server performances is obtained from a history log that is maintained by the RWS framework with recent invocations results. The selected server will be the one with the lowest response time for the most recent invocation involving the same requested operation with an equivalent set of parameters.

Best Median – This policy is a variation on the previous policy, where the selection is based on the $k$ most recent performances recorded for each server. The selected server will be the one with the lowest median response time amongst the set of past performances considered. Note that, for $k = 1$, Best Median is reduced to Best Last (in our experiments, $k$ was set to 6). The use of the median, instead of the mean, is due to the fact that the median is less affected by unusually large response times, which may occur sporadically, most notably at client peak times, due to the unpredictable nature of Internet latencies.

The next section reports on the results of applying those five policies to access a real-world replicated web service.

3. Empirical evaluation

3.1. Experiments

Our empirical evaluation was carried out from 12 to 31 December 2003, excluding weekends and public holydays. The experiments involved two client machines, both located at the city of Fortaleza, Brazil, dedicatedly running the same client application. That application used the RWS framework to invoke the
same replicated web service multiple times, in a way to apply each of the five server selection policies described previously. As the target replicated web service we used the service *UDDI Business Registry* [18], which is provided by four companies (namely Microsoft, IBM, SAP and NTT) located in three different continents: North America (Microsoft and IBM), Europe (SAP) and Asia (NTT). Figure 3 illustrates the geographic location of the two clients in Brazil and the four replicas in their respective continents.

Despite being located in the same city, the two client machines presented significant differences in terms of work environment and connection bandwidth. One client was located at the University of Fortaleza (referred to as client UNIFOR), and connected to the Internet through a government-run 2Mbps academic backbone. The other client was located at the IT department of a Brazilian bank (referred to as client BNB), and connected to the Internet through a 4Mbps commercial backbone.

Each replica of the service was invoked via the same operation, *GetServiceDetail*, which returns UDDI information about a list of services whose identification keys are provided as invocation parameters by the client application [18]. Since UDDI information structures are of very similar size, even for different services, by varying the number of identification keys passed as invocation parameters it was possible to accurately estimate the response size expected from each service invocation.

The client application was implemented in a way to continuously invoke the replicated service, so that we could observe how factors such as invocation period, response size, and server selection policy used would affect the perceived response time at both clients. To this end, all service invocations were organized in sessions of five cycles each. Each cycle consisted of a sequence of five service invocations, using the same set of parameters (i.e., the same list of identification keys), with each invocation using a different server selection policy. The numbers of identification keys used as invocation parameters in each of the five cycles of a session were 1, 10, 20, 30 and 40, respectively. Those numbers of keys corresponded to responses of size 9Kb, 87Kb, 174Kb, 260Kb and 347Kb, approximately.

In each cycle, a maximum invocation time was defined, within which every pending responses would have to be completely received. In the case that a full response failed to be received within the expected time, the corresponding invocation would be recorded as a *time-out* exception. This measure was necessary to avoid overly long invocations, which could prevent the execution of a minimum number of sessions each day. The maximum invocation time defined for each cycle was estimated based on the mean response time observed during a calibration period, in which the same set of experiments were run without imposing any time restriction on the duration of invocations.

### 3.2. Results

#### 3.2.1. Analysis method.

Just like past results are treated by the Best Median policy, response times observed during the experiments were analyzed considering the median as the measure of central tendency. The reason, again, is to minimize the impact of overly long network latencies observed during client peak times.

In our analysis, we investigated the effects of invocation time and response size on the observed mean response time for each server selection policy at both clients. In addition, we also computed the relative speedup offered by each policy with respect to Random Selection, which produced the worst overall performance amongst the five server selection policies implemented.

#### 3.2.2. Effects of invocation time.

Initially, we analyzed the performance of the five server selection policies with respect to invocation time. The aim was to identify those periods of the day when the observed response times presented the most significant variations. Figures 4 and 5 depict the results obtained at each client, considering the observed mean response time along each hour of the day. The graphics only show results relative to cycle 5, which, due to a large response size, presented the largest variations.

At client UNIFOR, we observed a significant increase in response time after 7am, with a corresponding decrease to the same levels only near 11pm. This period matches exactly the period of most intense academic activities at the university. We also
observed a high variability of performance throughout the day, even for those policies with the lowest response times. At 8am, for instance, the response time of some policies are already up to 6 seconds higher than the response time obtained by the same policies at 7am. The period of worst performances is around 3pm.

At client BNB, considering only those policies with the best performances, we can see that the period of higher response times is between 8am and 5pm, approximately. This period also matches the period of most intense activities at the bank. The periods of worst performances, for the best policies, are around 11am and 3pm.

In general, we observed that response times obtained at client BNB are at higher levels than those obtained at client UNIFOR. On the other hand, client BNB seems to be more scalable, as variations in response times observed at that client are of lower magnitude than those observed at client UNIFOR. At client BNB, the amplitude between the lowest and the highest points of the best performance curve is about 5 seconds, whereas at client UNIFOR this same amplitude is near 16 seconds.

It is interesting to note that the Random Selection policy presented the worst overall performance at both clients, independently of response size and invocation time. The reason is that the random selection process treats all replicas equally. This means that it may select momentarily slower servers as often as the fast-responding ones, which clearly degrades the policy’s overall performance. Our Random Selection results comply with those described in [9], which were obtained in the context of accessing traditional web resources such as images and HTML documents.

3.2.3. Relative speed up. To better visualize the performance differences amongst the five policies, we analyzed the relative speedup, in terms of median response time, offered by each of the other four policies with respect to Random Selection, which presented the worst overall performance amongst the five policies, at both clients.

Figure 6(a) depicts the relative speedup observed at client UNIFOR, during the periods of more intense activities (i.e., from 7am to 11pm). From that figure we can see that highest speedups were offered by the two policies based on historical data, Best Median and Best Last, followed by HTTPing and Parallel Invocation, in that order. The best case for Best Median was observed with 30-key invocations (Cycle 4), where that policy
produced a median response time up to 2.3 times faster than that observed with Random Selection. This result is justifiable, since both Best Median and Best Last consider recent invocation results as part of their selection process, thus avoiding momentarily slow servers. The negative results for Parallel Invocation is possibly due to the slower connection capacity of client UNIFOR, which may be insufficient to cope with the concurrency generated by the incoming traffic when the four servers are invoked simultaneously.

Figure 6(b) in turn shows the relative speedup observed at client BNB, also at the periods of more intense activities (in this case, from 8am to 5pm). In general, we can see that the overall speedup factor observed at this client is considerably lower than that observed at client UNIFOR, for the same policies. Another notable difference is in the results observed for Parallel Invocation. In contrast to the results observed at client UNIFOR, where Parallel Invocation offered the worst speedup, here that policy offered a speedup factor that is visibly higher than those of the other three policies. The speedup is more significant for 10 and 20-keys invocations (Cycles 2 and 3, respectively), with Best Last and Best Median, especially the latter, only catching up at the last two cycles. These results suggest that, contrary to what was observed at client UNIFOR, the connection capacity of client BNB was enough to handle the incoming traffic generated by parallel invocation.

The relative speedup offered by HTTPing was lower than that offered by Best Last and Best Median, at both clients. However, with regards to Parallel Invocation, the relative performance of HTTPing varied from client to client, being above it at client UNIFOR but below it at client BNB.

4. Related work

Despite being a relatively recent research area, dynamic selection and invocation of replicated (or functionally equivalent) web services is attracting an increasing interest from the research community. In this section, we compare three recently proposed works ([11][3][13]) that are more closely related to our work on the RWS framework.

In [3], Azevedo et al. propose a mechanism for dynamic selection of semantically equivalent web services. The selection mechanism is based on a number of quality and cost criteria, such as service initialization time, expected execution time for each service operation, monetary cost of service execution, and cost of compensation in case of failure. Based on those criteria, the authors propose three service selection strategies: best monetary cost, best performance (response time), and best tradeoff between cost and performance. Since we do not consider monetary costs in our work, here we only discuss the best performance strategy. With regards to that strategy, the authors simply suggest using parallel invocation as an ideal policy, without giving any empirical evidence to support that claim. In contrast to the work of Azevedo et al., we have implemented and empirically evaluated a variety of replica selection policies in a real-world Internet setting. In addition, contrary to those authors’ claim, we have shown that parallel invocation may not always be the ideal choice for server selection, and that the perceived service performance under that policy may depend on, amongst other factors, the capacity of the client connection at hand.

Parallel invocation is also used by the invocation mechanism proposed by Keidl et al. to access functionally equivalent web services [11]. In that work,
the authors propose three service invocation modes. In the first mode, a single service is selected at a time; in case of failure, another server is selected amongst the remaining equivalent services. In that mode, no server selection policy is explicitly used, with the servers being selected according to their position within the list of equivalent services returned by the UDDI registry. In the second mode, only part of the set of equivalent services is invoked in parallel (i.e., concurrently). A client-defined parameter is used to control the number of services actually invoked; as in the first mode, server selection is again non-deterministic, based on the order the services URLs are returned from the registry. Finally, in the third mode, the whole set of equivalent services is invoked in parallel. In the two modes involving parallel invocation there is a parameter that can be set by the application to allow completion of all pending responses, and not only the one that is received first. With this parameter enabled, the invocation mechanism will compare all responses received according to some quality criteria, and only return to the client application the one that best satisfies those criteria. The price for selecting the best response at the client side is that this may heavily affect the overall response time perceived by the application. In our work, in contrast, since all service replicas are considered to be identical, there is no need to compare or even retain any pending response while using the parallel invocation policy. By discarding all pending responses but the first, the overhead imposed by contacting all servers in parallel is minimized.

Padovitz et al. have also proposed mechanisms for the dynamic selection of web services [13]. That work, differently from our work and the works discussed above, focuses on ways to obtain accurate information on the state of each service, immediately prior to issuing an invocation. To this end, the authors propose a system to retrieve state information from service providers, according to client-defined properties, such as performance, availability, and quality of service. Based on the information received from each server, the system then selects the “best” server for invocation. Two information retrieval approaches are proposed. One based on parallel invocation of service providers via RPC, and the other based on parallel or circular dispatching of mobile agents to the services’ remote hosts. In either case, the server selection process only takes into account information directly provided the servers themselves. In our work, in contrast, server selection is based on server information as it is perceived by each client application. Another negative aspect of the work of Padovitz et al. is that, since the system is still under development (as reported on in [13]), the authors fail to provide any empirical evidence of the potential benefits of their approach.

5. Conclusion

This work presented the RWS framework for dynamic invocation of geographically replicated web services. RWS supports different server selection policies, in a transparent way, so as to improve service access time at the client side. Five server selection policies were implemented and empirically evaluated in a real-world Internet setting, namely Random Selection, Parallel Invocation, HTTPing, Best Last, and Best Median. Our main experimental results are summarized below.

The two server selection policies based on historical data, Best Last and Best Median, presented the best performance overall, with very similar results at both clients. Therefore, they may offer the best invocation alternative when clients characteristics are not taken into account. However, more experiments are necessary to determine whether one of the two policies (or even another variation of Best Median, with a different k value) may produce results significantly better than the other. It should be emphasized, though, that the performance of any server selection policy based on historical data is highly dependent on the existence of recent data about the state of each server. In our experiments, this issue was addressed by invoking the four servers, individually, at the beginning of each cycle. A more realistic solution would require periodic monitoring of each server, possibly involving components external to the RWS framework.

The Parallel Invocation policy presented the best results at the client whose connection characteristics were enough to handle the concurrency generated when receiving multiple responses simultaneously. On the other hand, at the client where the connection was insufficient to handle the concurrency of parallel invocation, the perceived performance gradually degraded as the response size increased. Nevertheless, our experiments involved only four servers. Further experiments are necessary to determine from which point the relation between the available connection bandwidth and the concurrency level generated by parallel invocation may land that policy prohibitive.

The HTTPing policy presented an intermediate performance level at both clients, always better than Random Selection, the worst server selection policy overall, but also always worse than Best Median and Best Last.

It is worthy noting that, in the context of related works on the evaluation of server selection policies for accessing traditional web resources, the observed server response time only took into account factors such as network latency, content request and retrieval
time, and, for those policies based on historical data, server performance and workload. In our experiments, in addition to all those factors, the observed service response time also included the XML processing overhead imposed by the SOAP protocol at both the client and the server sides. This overhead involved, amongst other costs, the time to serialize (de-serialize) the SOAP messages, in the form of XML documents, at both sides, and also the time necessary for the application server to dispatch and invoke the requested web service operation, at the server side.

As regards future work, some interesting lines of research include expanding the experiments to new services and new client settings; designing new server selection policies, possibly from the combination of those five described here; re-implementing (and re-evaluating) the RWS framework in the form of a proxy service; and devising mechanisms for automatically updating replica location information, possibly benefiting from the notification service provided by the new version (3.0) of the UDDI Business Registry specification [4].

6. References


