Client-side selection of replicated web services:
An empirical assessment

Nabor C. Mendonça a,*, José Airton F. Silva a, Ricardo O. Anido b,1

a Mestrado em Informática Aplicada (MIA), Universidade de Fortaleza (UNIFOR), Av. Washington Soares, 1321, 60811-905 Fortaleza, CE, Brazil
b Instituto de Computação (IC), Universidade Estadual de Campinas (UNICAMP), Caixa Postal 6176, 13084-971 Campinas, SP, Brazil

Received 2 March 2007; received in revised form 23 October 2007; accepted 1 November 2007
Available online 17 November 2007

Abstract

Replicating web services over physically distributed servers can offer client applications a number of QoS benefits, including higher availability and reduced response time. However, selecting the “best” service replica to invoke at the client-side is not a trivial task, as this requires taking into account factors such as local and external network conditions, and the servers’ current workload. This paper presents an empirical assessment of five representative client-side service selection policies for accessing replicated web services. The assessment measured the response time obtained with each of the five policies, at two different client configurations, when accessing a world-wide replicated service with four replicas located in three continents. The assessment’s results were analyzed both quantitatively and qualitatively. In essence, the results show that, in addition to the QoS levels provided by the external network and the remote servers, characteristics of the local client environment can have a significant impact on the performance of some of the policies investigated. In this regard, the paper presents a set of guidelines to help application developers in identifying a server selection policy that best suits a particular service replication scenario.

Keywords: Web service replication; Web service selection; Empirical evaluation

1. Introduction

Web services are becoming the de facto standard for software application development and deployment based on the emerging Service Oriented Computing (SOC) paradigm (Papazoglou and Georgakopoulos, 2003). Along with their associated technologies – such as SOAP (W3C, 2003), a message exchange protocol for service interactions; WSDL (W3C, 2001), a language for describing service interfaces; and UDDI (OASIS, 2002), a repository for dynamic service publication and discovery – web services offer a powerful mechanism for integrating existing software applications over the web, independently of programming language, execution platform or transport protocol (Cauldwell et al., 2001).

However, to be used effectively in real-world application scenarios, future web services technologies will have to provide application developers with important Quality of Service (QoS) attributes, such as efficiency, availability, reliability and security (Menasce, 2002). Efficiency and availability, in particular, can be provided through the use of service replication (Salas et al., 2006). Specifically, replication has the potential to increase service availability, by allowing a service to be provided redundantly at multiple physically distributed servers; replication can also improve service response time, by allowing client applications to invoke the service from a less loaded or topologically closer server.

Traditionally, two main approaches have been used to implement efficient access to replicated resources over the
web. The first approach aims at increasing system throughput at the server side, by automatically re-distributing client requests among a set of cooperating servers. A typical use of this approach is when resources are transparently replicated over a cluster of servers of the same network domain (e.g. Damani et al., 1997). The second approach aims at improving QoS guarantees at the client-side (notably performance and high availability), by exploring “smart” server selection mechanisms that can transparently select the best server to invoke on behalf of each client application, based on client-specific QoS demands (Yoshikawa et al., 1997; Sayal et al., 1998; Dykes et al., 2000; Hanna et al., 2001; Amini et al., 2003). This approach is therefore more appropriate for wide-area replication scenarios, in which resources are replicated over servers that are geographically or administratively apart from each other (Salas et al., 2006).

Our work on web service replication follows the latter approach. Specifically, we are interested in addressing the challenges of automatically selecting the best web service (from the perspective of a particular client application) amongst a collection of functionally equivalent web service instances. The study of efficient selection strategies for accessing replicated or functionally equivalent web services has attracted a growing attention from the web services research community in recent years (e.g. Padovitz et al., 2003; Keidl and Kemper, 2004; Costa et al., 2004; Tian et al., 2004; Liu et al., 2004; Yu and Lin, 2005a; Hu et al., 2005; Makris et al., 2006; Salas et al., 2006).

However, most of the service selection solutions proposed thus far suffer from one or more of the following limitations:

- they require modifications to the standard SOC model, often by extending the existing SOC technologies, such as WSDL and UDDI, with novel mechanisms to support publication and discovery of QoS service information (e.g. Tian et al., 2004; Liu et al., 2004; Yu and Lin, 2005a; Hu et al., 2005; Makris et al., 2006). This means that their solutions might not be readily applied to existing service-oriented applications, unless those applications are modified to comply with their proposed extended model;
- they tend to neglect service response time—as it is perceived at the client-side—as a first-class QoS attribute (e.g. Padovitz et al., 2003; Keidl and Kemper, 2004); or
- they simply fail to provide any empirical evidence of the performance gains provided by their service selection strategies when applied to world-wide replication scenarios—one notable exception is the work of Salas et al. (2006).

Motivated by the above limitations, in this paper, we present an empirical assessment of the performance impact of five representative server selection policies, from the perspective of two different client configurations, when used to access a world-wide replicated web service. The assessment involved two client machines, with significant differences in terms of connection and workload characteristics, continuously selecting and invoking one of the four replicas of the target replicated service, using each of the five server selection policies at a time, for a period of three consecutive weeks. The response times obtained with each policy at each client machine were then analyzed both quantitatively and qualitatively.

Our analysis shows that, in addition to the QoS attributes of the external network and the remote service providers, characteristics of the local client environment can have a significant impact on the performance of some of the server selection policies investigated. This finding is important in that it suggests that the performance impact (either positive or negative) of a given server selection strategy may not be possible to estimate a priori without taking into account the characteristics of the client application scenario at hand. In this regard, we have formulated a set of practical guidelines, based on our experimental results, to help application developers in identifying a server selection policy that best suits the needs of a particular service replication scenario.

We believe that our results could also be useful to developers of service-oriented technologies. In particular, web service middleware developers may find our results useful as a stepping stone upon which to design and experiment with new, more effective server selection strategies, possibly by building on our own policies.

This paper is a revised and substantially extended version of a previous conference paper (Mendonça and Silva, 2005). The new contributions include a more detailed description of the five server selection policies investigated (Section 2); a more in-depth analysis of the results, including a novel quantitative evaluation based on the policies’ observed server selection decisions (Section 3); and a discussion of our work’s main limitations and their implications to future research in the area (Section 4). The paper also covers related work in Section 5. Finally, the paper summarizes our results and guidelines, and outlines possible directions for future work (Section 6).

2. Server selection policies

The five replica selection policies investigated in our work are: random selection, parallel invocation, HTTPing, best last and best medium. These policies were chosen because they are representative of the wide range of server selection techniques that have already been studied in the context of accessing more traditional replicated web resources (e.g. Sayal et al., 1998; Dykes et al., 2000; Hanna et al., 2001; Amini et al., 2003).

The following subsections describe the five policies in terms of their underlying principles and potential performance and load balance implications. The section ends with some details on their implementation as part of a
general server selection mechanism that we have integrated into an existing web services development framework.

2.1. Random selection

As the name implies, this policy selects the server to be invoked randomly amongst the set of replicated servers. If the selected server fails to respond, another server will be selected amongst the set of remaining servers, and so on, until no more servers are left for selection, in which case an invocation exception will be returned to the client application.

The consequence of using this policy is that clients will treat servers equally, selecting each of them with the same probability. From a load balance perspective, random selection is a fair policy, since it distributes client requests evenly amongst the set of replicated servers.

Random selection was one of several server selection strategies investigated by Dykes et al. (2000), but in the context of accessing traditional web resources, such as HTML documents and image files.

2.2. Parallel invocation

This policy invokes all servers “in parallel” (in practice, the replicas are invoked concurrently, using threads). The first response to be received in full is returned to the client application; all pending invocations are then interrupted, with their partial response, if any, being discarded. Only in the case that all servers fail to respond, an invocation exception will be returned to the application.

Compared to random selection, the main benefit of the parallel invocation policy is that clients will not need to wait until a server communication failure is detected by the underlying network before invoking an alternative server. On the other hand, if two or more servers respond to the invocation requests sent in parallel, as it is likely to happen in most cases, their responses will compete for local network resources at each client. This means that the parallel invocation policy may impose a negative performance impact on clients, if their local network resources are unable to cope with the servers’ concurrent traffic. From a load balance perspective, parallel invocation is a highly ineffective policy, since it always propagates client requests to all servers.

The parallel invocation policy is an adaptation of the dynamic parallel access method proposed by Rodriguez and Biersack (2002) to download replicated Internet files. Variations on this policy were also investigated in the context of accessing functionally equivalent web services by Keidl and Kemper (2004) and Costa et al. (2004).

2.3. HTTPing

Like the parallel invocation policy, HTTPing also attempts to contact all servers in parallel, but only by means of a small “probe” message in the form of a HTTP HEAD request.\(^2\) The first server to respond to the probe is then selected for invocation. If that server fails to respond, the server with the second fastest probe response is selected, and so on, until no more servers are left for selection, in which case an invocation exception is returned to the client application.

The consequence of using the HTTPing policy is that clients will favor only those servers that are faster to respond to the probe. This means that the quality of selection obtained with this policy will depend directly on how accurate the probe mechanism is in predicting service performance at invocation time. From a load balance perspective, HTTPing offers a compromise between the random selection and parallel invocation policies. Like parallel invocation, HTTPing contacts all servers in parallel at every invocation; however, since the group of servers is contacted using only a small probe message, with only one server actually being invoked at a time, the concurrency overhead generated by this policy is much lower. On the other hand, this policy will concentrate the load on a single server as long as that server is the fastest to respond to the probe.

Because the probe mechanism used by HTTPing is totally transparent to the application programmer (as we will explain in Section 2.6), the overhead of sending the probes and then waiting for their responses will be included as part of the overall service response time, as it is perceived at the client-side. One possible way to minimize this overhead is by sending probes sporadically, instead of prior to every service invocation. This approach was first proposed by Hanna et al. (2001). However, even this approach would be ineffective if the probe response times could vary in unpredictable ways.

The HTTPing policy is a variation on the probe-based serve selection strategy proposed by Dykes et al. (2000). In that work, a TCP-level probe mechanism was used for server selection in the context of accessing traditional web resources. Compared to the work of Dykes et al., our solution has the advantage of taking into account the servers’ current workload, in addition to their TCP round trip latency. Another benefit of our solution is its increased flexibility, given that, for security reasons, many web servers are now configured to refuse or ignore TCP-level probe requests.

2.4. Best last

This policy selects the best server to be invoked based on past performance results. Specifically, it selects the server with the best recent performance, considering only equivalent successful invocations issued to the same service.\(^3\)

\(^2\) The size of a HTTP HEAD request message is typically in the order of a few hundreds of bytes — hence the name HTTPing.

\(^3\) In our work, we consider that two service invocations are equivalent when they invoke the same service operation with similar sets of parameters, and produce responses of similar sizes.
Information on past invocation results is obtained from a historical invocation log that is maintained by the server selection mechanism. The log records information about past successful service invocations, such as invocation time, server identification, operation name, request parameters and response size. The actual number of entries that need to be kept in the log will depend on factors such as the frequency upon which new service invocations are issued by the client application, the expected overall service response time, and the expiration time for past invocation results (i.e., the amount of time after which a certain past invocation result can no longer be considered valid for estimating a server’s future performance).

The consequence of using the best last policy is that the selection process will keep selecting the same server as long as that server produces better results than those results previously recorded for the other servers. From a load balance perspective, this policy will also concentrate the load on a single server, as long as no other server with a better response time is invoked.

The quality of selection obtained with the best last policy will be highly dependent on how frequent invocations involving equivalent sets of parameters are issued to each server. The reason is that infrequent invocations may result in a historical invocation log that does not accurately reflect the servers’ current performance level or the network’s current workload condition. Quality of selection for the best last policy will also depend on how stable the performance of each server and the network is between service invocations. Note that any abrupt variation in any of these parameters is likely to compromise the accuracy of using past invocation information to predict future server behavior.

2.5. Best median

This policy generalizes the best last policy, by selecting the server with the best median performance among the \( k \) last successful invocations issued to each server. As with best last, when computing the median we consider only equivalent past invocations issued to the same server. The use of the median as an indication of central performance, instead of the mean, is justified by the fact that the mean is less affected by overly large response times, which may occur sporadically due to the unpredictable nature of Internet latencies (Dykes et al., 2000).

Clearly, the quality of the selection obtained with this policy will be affected strongly by the number of previous invocation results that are included in the computation of the median (the parameter \( k \)). For example, if \( k \) is too large, the median computation will include invocations that may be too old to reflect the servers’ current state. On the other hand, if \( k \) is too small, the selection mechanism may become too sensitive to momentary bad results. As a guideline, we suggest to set \( k \) to the maximum possible value that still captures past invocation results that are within a given threshold (say 20%) of each other. In practice, estimating an adequate value for \( k \) requires taking into account factors such as the frequency upon which the replicated service is invoked, and the relative stability of the servers’ performance and the network load between invocations. Note that, for \( k = 1 \), the best median policy behaves exactly like the best last policy.

A consequence of using the best median policy is that client applications will favor the same server, as long as no other servers are invoked that provide a better median response time. From a load balance perspective, this policy behaves similarly to the best last policy, i.e., it will concentrate the load on a single server as long as that server provides the best past performance results according to the policy’s selection criteria.

A notable difference between best last and best median is that the former will be more sensitive to momentary performance variations. For instance, with best last, a single bad performance result suffices for this policy to change its selection to a different server, while with best median there has to be a series of bad results for the same server (long enough so that its median performance is affected) before that policy changes its selection to another server.

2.6. Implementation details

We have implemented our five server selection policies in the form of a general server selection mechanism, which has been incorporated into the AXIS framework (Apache, 2006). AXIS is an open source web services development framework for the Java programming language. Integrating the new server selection mechanism with AXIS required extending the framework’s client-side infrastructure with new components, namely Selection Manager and Invocation Manager, as shown in Fig. 1.

The Selection Manager abstracts away details of the server selection process implementation from other AXIS components. In addition, this component provides a template for the implementation of new server selection policies. The Invocation Manager is responsible for carrying out the actual service invocation. One of its main functionalities is to create and coordinate the multiple execution threads necessary to implement the parallel invocation and HTTPing policies. It is implemented on top of AXIS’s native service invocation API.

![Fig. 1. AXIS-based implementation of the server selection mechanism.](image-url)
In order for AXIS to use our server selection components, we also had to extend its code generation tool, so that the generated client code (or stub) could invoke the target web service by means of a particular server selection policy, instead of using AXIS’s native invocation API directly. The stub code generated this way allows client application programmers either to specify the service replica to be invoked explicitly, or to delegate that decision to the server selection mechanism, based on a particular server selection policy. In the former case, the invocation process is carried out using AXIS’s original invocation API, while in the latter case the selection mechanism selects the best service replica to invoke on behalf of the requesting client application, according to the selection policy defined by the user.

We must note that, even though our implementation is currently tailored for AXIS, its modular design makes it easy to port to other Java and non-Java-based web service development frameworks, such as JWSDP (SUN, 2006) and .NET (Microsoft, 2006).

3. Empirical assessment

In this section, we describe our assessment methodology and analyze our results both quantitatively and qualitatively.

3.1. Methodology

3.1.1. Clients and servers

The assessment was carried out during three consecutive weeks (excluding weekends and public holidays) in December 2003. Our experiments consisted of two client machines, both located in the city of Fortaleza, Brazil, running the same client application in dedicate mode. Although physically located in the same geographic region, the two client machines presented significant differences in terms of hosting organization, workload distribution, and network capacity. One client machine (henceforth referred to as client UNIFOR) was based at the Computer Science Lab of the University of Fortaleza. At the time of the experiments, UNIFOR was connected to the Internet through a governmental 2 Mbps backbone. The other client machine (henceforth referred to as client BNB) was based at the IT department of a Brazilian bank. At the time of the experiments, BNB was connected to the Internet through a commercial 4 Mbps backbone. The configurations of the two client machines in terms of their hardware, operating system and network characteristics are summarized in Table 1.

The client application was implemented using our modified version of the AXIS framework. Apart from collecting and storing the experimental results, the application’s primary functionality was to continuously invoke one of the replicas of the UDDI Business Registry (UBR) web service (OASIS, 2002), using each of our five server selection policies at a time. The reason for choosing UBR was that, at the time of the experiments, it was the only publicly available web service replicated world-wide. UBR replicas were provided by four different IT companies in three separate continents: two in North America (USA), provided by Microsoft and IBM, respectively; one in Europe (Germany), provided by SAP; and one in Asia (Japan), provided by NTT. As of January 12, 2006, Microsoft, IBM and SAP (NTT was no longer a UBR provider) decided to discontinue their public UBR service. The full announcement is available at http://uddi.microsoft.com/about/FAQshutdown.htm.

### Table 1

<table>
<thead>
<tr>
<th>Client</th>
<th>Configuration</th>
<th>ISP</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIFOR</td>
<td>Pentium III 700 MHz, 192 MB RAM, Gov.</td>
<td>2 Mbps</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Windows 2000 Prof.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNB</td>
<td>Pentium III 800 MHz, 256 MB RAM, Comm.</td>
<td>4 Mbps</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Windows 2000 Server</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the case of the two replicas located in the US, we used publicly available IP location software to find out where exactly in the US they might have been deployed. According to the IP location systems used, the Microsoft replica was located in Redmond, Washington, while the IBM replica was located in Lake Mary, Florida. Fig. 2 shows the geographic location of the two client machines in Fortaleza, Brazil, and the four UBR service replicas in their respective continents.

3.1.2. Sessions and cycles

To facilitate data gathering and analysis, we organized all service invocations issued by each client application into sessions, with the invocations of each session being further organized into five subsequent cycles. Each cycle in turn consisted of a sequence of five identical invocations to the GetServiceDetail operation provided by UBR, with each invocation using a different server selection policy.

According to the UDDI specification (version 2.0) (OASIS, 2002), the GetServiceDetail operation returns a list of service properties registered at the target UDDI server for a list of services whose registration keys are passed as invocation parameters. Because all service properties are very similar in both structure and size, even for different services, by varying the number of registration keys passed as invocation parameters to that operation we were able to control precisely the size of the SOAP messages exchanged...
between the client application and the UBR service. This allowed us to run the experiments under a wide range of client-service interactions scenarios, involving the transmission of SOAP messages with a wide range of sizes. Specifically, the numbers of invocation parameters (i.e., service registration keys) used in each cycle were 1, 10, 20, 30, and 40, respectively. For successful invocations, those numbers generated SOAP response messages of size 9 Kbytes, 87 Kbytes, 174 Kbytes, 260 Kbytes and 347 Kbytes, respectively.

For each cycle, a maximum invocation time (MIT) was defined, within which every pending responses had to be completely received. When a full response was not received within the expected time, the corresponding invocation was recorded as a timeout exception. This measure was necessary to avoid overly long invocations, which inevitably occurred at network congestion times, and that could prevent the execution of a minimum number of sessions at each day of the assessment. The MIT value defined for each cycle was estimated based on the mean response time observed during a calibration period, prior to the experiments, in which similar experiments were run without imposing any time restriction on the duration of service invocations. This calibration period was also used to estimate a value for the parameter $k$ of the best median policy, which has been set to 6. In our experiments, this value represented a time window of approximately one hour of recorded past invocation results. This means that only previous invocations that occurred within that time frame were considered to compute the median.

### Table 2

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Invocation parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># Req. keys</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
</tr>
</tbody>
</table>

Here we analyze the availability and performance levels measured for the four UBR servers at each client machine, and the effects of both time of day and message size on the performance of the five server selection policies investigated.

#### 3.2. Quantitative results

Here we analyze the availability and performance levels measured for the four UBR servers at each client machine, and the effects of both time of day and message size on the performance of the five server selection policies investigated.

##### 3.2.1. Service availability

We observed a high level of availability (>99%) for the replicated UBR service at both clients, throughout the whole assessment. Table 3 presents a summary of the number of sessions and service invocations executed at each client, along with their corresponding success rate and number of failures. Out of a total of 38,511 invocations issued to the replicated service at client UNIFOR, only 312 could not be completed successfully, of which 165 were due to timeout failures and 147 were due to other failures. This represents a success rate of 99.19%. An even higher availability level was observed at client BNB. Out of a total of 50,985 invocations issued to the replicated service at that client, only 148 could not be completed successfully, of which 65 were due to timeout failures and 83 were due to other failures. This represents a success rate of 99.71%.

Note that we were able to execute roughly 30% more sessions at client BNB than at client UNIFOR. This difference reflects the fact that we observed smaller response times at client BNB, which resulted in sessions with shorter durations at that client. We discuss the servers’ individual performances in the next subsection.

##### 3.2.2. Servers performance

To establish a basis upon which to assess the performance of the five server selection policies investigated in our experiments, we first characterize the performance of the four servers individually. To this end, we calculated the median response time obtained with each server in each session cycle, at both client machines. The results are shown in Fig. 3.

At client UNIFOR, with invocations involving only one message element (cycle 1) the four servers present very similar performance levels. As the message size grows, the servers present quite different performance results, with the best response times obtained with Microsoft, IBM, SAP and NTT, in that order. Note that for invocations involving

### Table 3

<table>
<thead>
<tr>
<th>Client</th>
<th># Sessions</th>
<th># Service invocations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>UNIFOR</td>
<td>861</td>
<td>38,511</td>
</tr>
<tr>
<td>BNB</td>
<td>1133</td>
<td>50,985</td>
</tr>
</tbody>
</table>
larger messages (cycles 4 and 5), SAP and NTT distance themselves from the other two serves, with both generating median response times nearly three times that of Microsoft.

At client BNB, we observe that Microsoft performs slightly better than IBM at cycles 1, 2 and 3, with IBM finally catching up and then surpassing Microsoft at cycles 4 and 5, respectively. NTT and SAP follow behind, in that order, with their performance levels (particularly the latter’s) gradually departing from that of Microsoft and IBM as the message size grows.

When we look at the results of both clients, we can see that IBM and SAP present very similar performance fluctuations across all cycles. The same does not happen to Microsoft and NTT. While Microsoft’s median response time ranges from around 4 s (cycle 1) to 13 s (cycle 5) at UNIFOR, at BNB it degrades much more quickly, ranging from around 3 s (cycle 1) to 21 s (cycle 5). The opposite effect is observed for NTT. At UNIFOR, NTT’s median response time quickly degrades from around 5 s (cycle 1) to 35 s (cycle 5), while at BNB its degradation is much slower, ranging from around 5 s (cycle 1) to 25 s (cycle 5).

Overall, we can classify the performance of the four servers, at both clients, in the following order: Microsoft, IBM, NTT and SAP. Servers Microsoft and IBM presented the best response times in all cycles, with a slight advantage to the former. Servers NTT and SAP presented the worst results, especially in cycles 4 and 5, which involved the largest messages. SAP performance in those cycles is particularly bad at client BNB, with a median response time nearly 10 s above that of NTT in cycle 5.

3.2.3. Time-of-day effects

Here we analyze the influence of time of day on the performance of the five server selection policies. The aim is to identify those periods in which the observed response times presented the most significant variations. Fig. 4 depicts the results obtained at each client machine, considering the mean response time observed with each policy along the 24 h of day. The graphics only show results relative to cycle 5, which, due to a larger response message, presented the widest variations.

At client UNIFOR, we observe a significant increase in the replicated service response time after 7 am, with a corresponding decrease to the same levels only near 11 pm. 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 8 am, for instance, the response times observed for some policies are already up to 6 s higher than the response times obtained with the same policies at 7 am. The period of worst performances is around 3 pm.

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

Overall, 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 fluctuations in response times observed at that client are of lower magnitude than those observed at client UNIFOR. At client BNB, for instance, the amplitude (i.e. the absolute difference between the lowest point and the highest point) of the performance curve for the best policy (i.e. parallel invocation) is about 5 s, whereas at client UNIFOR the curve amplitude for best policy (i.e. best median) is near 16 s.

It is interesting to note that the random selection policy presented the worst performance results at both clients, independently of response size and time of day. This is justifiable, given that, even though the four servers presented very different performance levels at both clients, as we have discussed previously, random selection always treats all servers equally. This means that it is likely to select the slower servers (i.e. NTT and SAP) as often as the faster ones (i.e. Microsoft and IBM), which, therefore, will degrade its overall performance.

Our results for random selection are in accordance with those described by Dykes et al. (2000), which were obtained in the context of accessing replicated web content in the form of images files and HTML documents.

3.2.4. Relative speedup

We now analyze the relative speedup, in terms of the median response time, offered by each of the other four policies with respect to random selection, during the periods of more intense activities at both clients (i.e., from 7 am to 11 pm at client UNIFOR, and from 8 am to 5 pm at client BNB). We chose random selection as our comparison baseline because that policy presented the worst performance results overall. The speedup analysis is meant to better illustrate the performance differences observed amongst the five policies at the two client machines. The analysis results are shown in Fig. 5.

At client UNIFOR, the highest speedups were offered by the two policies based on past invocation results (i.e. 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 generated 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 slower servers. The negative result for parallel invocation, on the other hand, was somewhat of a surprise, given that policy was found to be amongst the best policies at client BNB (see below). A closer analysis of the parallel invocation results at client UNIFOR later revealed that its slower connection speed was insufficient to cope with the high concurrency generated by the incoming traffic when the four servers are invoked in parallel, thus degrading its overall performance.

At client BNB, we can see that the overall speedup factors observed for the four policies is considerably lower than those observed at client UNIFOR. 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 offers 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 bandwidth available at client BNB was enough to handle the intense incoming traffic generated by parallel invocation.

The relative speedup offered by HTTPing is at an intermediary level at client UNIFOR, where it is higher than parallel invocation but below best last and best median. However, it presents the worst speedup at client BNB, where it is only above random selection in absolute terms. The reason for such weak performance was due to the low quality of its selection strategy. A more detailed analysis of the quality of selection produced by HTTPing and the other policies will be presented in Section 3.3.

3.2.5. Cumulative distribution

Even though the median response time gives a good indication of the average performance of each policy, it offers no insight into the whole spectrum of individual response times observed for the five policies at the two clients. To address this issue, we have computed the cumulative distribution function over the total of response times obtained with each policy. The cumulative distribution function makes it possible to evaluate, in a more general perspective, the performance of the five policies in terms of response times intervals, as they appeared throughout the whole empirical assessment. Fig. 6 shows the cumulative distribution results for each of the five policies at the two clients. As with the speedup analysis, the figure only shows results relative to cycle 5, which involved the largest response messages.

The cumulative distribution analysis allows us to compare the five server selection policies within specific performance levels. For example, we can see from Fig. 6 that 50% of all response times obtained with the best last and best median policies at client UNIFOR were equal to or below 11.5 s, whereas the third best policy at that client, HTTPing, had only 35% of all of its observed response times within that interval. Similarly, 50% of all response times obtained with the best median and parallel invocation policies at client BNB were equal to or below 19 s, whereas best last and HTTPing had only 45% and 35% of all their response times, respectively, within that interval.

The fact that we observed a better response time distribution for most of the policies at client UNIFOR, despite the fact that BNB had a more robust network connection, is justified since their performances are largely dominated by the performance of the Microsoft server. As we have described in Section 3.2.2, Microsoft’s performance is less stable at BNB, where it degrades much more quickly than at UNIFOR.
3.3. Qualitative results

We now analyze the quality of selection produced by four of the five server selection policies investigated: parallel invocation, HTTPing, best last and best median. Random selection was deliberately left out of the analysis because, by definition, it always selects each server with the same probability.

Our analysis consisted of computing the ratio which each of the four policies selected each of the four UBR servers along the experiments, and by correlating those results with the performance results observed for each server individually. This allowed us to perform a more quality-based evaluation of the four policies, in terms of their ability to concentrate their selection on those servers which were known to offer the best performance results. We also discuss the influence that the characteristics of the replicated servers and the external network might have had on our results.

3.3.1. Parallel invocation

Fig. 7 depicts the selection ratio for the four UBR servers, as observed for the parallel invocation policy. We can see that the highest selection ratio observed at client UNIFOR was for the Microsoft server, which was selected in nearly 90% of all invocations executed in that client, followed far behind by IBM (with approximately 10% of the invocations), SAP and NTT, with the latter two servers being virtually ignored.

The server selection ratios observed at client BNB were largely influenced by message size, and showed significant differences across the five cycles. In cycle 1, which involved the smallest messages, Microsoft was the only server selected in all invocations, with the other three servers being completely ignored. As the message size grows, the selection ratio for Microsoft rapidly decreases at the same time that the selection ratio for IBM increases, with the latter finally beating the former (with 49% and 42% of the invocations, respectively) in cycle 5. NTT is selected in nearly 10% of the invocations in cycles 4 and 5. SAP, again, is virtually ignored in all cycles.

Overall, we can say that variations on the servers’ selection ratios for parallel invocation directly reflect the performance variations observed for the four servers at each client (see Section 3.2.2), which resulted in its selection process being prominently dominated by the two most responsive servers, Microsoft and IBM.

With respect to quality of selection, even though the parallel invocation policy selected the two servers with the best individual performances (i.e. Microsoft and IBM) in most of the invocations executed at the two clients, its overall performance showed significant variations from one client to the other. As we have described in Section 3.2.4, parallel invocation produced the best performance

---

Fig. 6. Cumulative distribution of individual response times observed for the five server selection policies in cycle 5: UNIFOR (top) and BNB (bottom).
results at client BNB, but was the second worst policy at client UNIFOR. Those differences were due, to a great extent, to the lower connection bandwidth available at client UNIFOR. This turned out to be insufficient to cope with the incoming concurrent traffic generated by invoking the four servers in parallel.

The implication of this finding to our study is that server selection ratio alone cannot be considered as a good metric to predict policy behavior when invocations are executed in parallel. This is because it fails to account for the concurrency overhead which that policy might generate at the client-side. In such scenarios, it is also important to consider the invocations’ expected message traffic (which in turn is influenced by both message size and the number of servers that are invoked concurrently) in light of the available connection bandwidth.

3.3.2. HTTPing

The server selection ratios observed for the HTTPing policy are shown in Fig. 8. Again, Microsoft was by far the most frequently selected server at client UNIFOR, with selection ratios ranging from around 60%, in cycle 1, to 80%, in cycle 5. SAP followed as the second most selected server, with selection ratios ranging from nearly 30%, in cycle 1, to 18%, in cycle 5. IBM and NTT were virtually ignored by HTTPing at that client.

Following far behind was IBM, with selection ratios in the order of 25%. NTT and SAP were virtually ignored by HTTPing at that client.

Differently from parallel invocation, the server selection ratios observed for HTTPing did not directly reflect the servers’ individual performances. One possible explanation is that, with HTTPing, the selection process is not dictated by the time each server takes to respond to a service invocation (which response messages gradually increased from cycle 1 to cycle 5), but by the time they take to respond to the HTTP probe (which is much smaller than a service response message and remained constant in size across all cycles).

In terms of quality of selection, HTTPing, like the parallel invocation policy, concentrated most of its invocations on the server with the best median response time (i.e. Microsoft), at both clients. Even so, that policy presented varying performance results in the two clients, being only the third and the fourth best policy (amongst the five policies investigated) at client UNIFOR and client BNB, respectively. The main reason for such poor performance is that the first server to respond to the HTTP probe (and which was selected for invocation by HTTPing) in many occasions was not the fastest server to return a complete service response to the client application. This was the case of SAP, at client UNIFOR. While this server, along with NTT, presented the worst individual performance at that client, with a median response time nearly three times that of Microsoft in cycle 5,
it was the second most selected server by HTTPing, with a selection ratio of about 18% in that cycle. A similar result could be observed at client BNB, where HTTPing clearly favoured the Microsoft server in its selections, with a selection ratio above 70% in all cycles, even though the IBM server offered an equivalent (and sometimes even higher) performance level at that same client.

The reason for such discrepancy between probe response time and service response time is that longer messages (such as those produced by the UBR service in cycles 4 and 5) tend to be more strongly affected by periods of network overload than smaller ones (such as the HTTP HEAD probes). Therefore, even though a server could be one of the first servers to respond to the probe, this would not necessarily mean that the same server would, when invoked, be the first to send a complete SOAP response to the client application.

Another factor that might have contributed to the weak performance observed with HTTPing was that, to be fair with the other policies, the time taken to send the probe and wait for its first response had to be included when computing the service’s overall invocation time. However, even when we exclude the probe overhead from our analysis, the HTTPing performance still suffers from bad selection choices, as it was demonstrated with the selection of the SAP server at client UNIFOR.

Our findings with respect to the HTTPing policy contrast with the probe results reported by Dykes et al. (2000), where a TCP-level probe mechanism was found to be the best server selection strategy for accessing traditional web resources. The reason seems to be that our experiments involved a service whose responses are relatively larger compared to the size of the HTTP HEAD messages used as the probe. Since larger messages are more likely to be affected by external network conditions, the use of the probe mechanism turned out to be an ineffective means of predicting service response time.

3.3.3. Best last and best median

Figs. 9 and 10 depict the server selection ratios observed for the best last and best median policies, respectively. We can see that the two policies produced results very similar to those observed for parallel invocation at client UNIFOR. In the case of best last, the Microsoft server dominated most of the invocations at that client, accounting for a select ratio around 90% in all cycles. The IBM server followed far behind, with a selection ratio in the range of 5–10%. NTT and SAP were mostly ignored by the two policies. In the case of best median, one notable difference was that the Microsoft server reached a selection ratio close to 100% in all cycles, with the other three servers being virtually ignored.

We also observed selection ratios for the two policies similar to those obtained for parallel invocation at client BNB, where the Microsoft server dominates most of the selections in cycles 1, 2 and 3, with the IBM server gradually catching up until it dominates the selections in cycle 4 (in the case of best median) or cycle 5 (in the case of best last).
Fig. 9. Server selection results for best last: UNIFOR (top) and BNB (bottom).

Fig. 10. Server selection results for best median: UNIFOR (top) and BNB (bottom).
It is important to note that, although the three policies (best last, best median and parallel invocation) presented similar quality of selection results at both clients, their performances varied considerably from one client to the other, as we have reported in Section 3.2. Therefore, it is reasonable to conclude that variations on their performance cannot be analyzed in terms of quality of selection alone. In fact, as we have pointed out in Section 3.2.4, the reason for parallel invocation to have performed at the same level of best last and best median at client BNB, but at a much lower level at client UNIFOR, was that the network bandwidth available at client UNIFOR was insufficient to cope with the incoming traffic generated when the four UBR servers were invoked in parallel. The effect of the local network capacity on the performance of parallel invocation at client UNIFOR is also visible in the fact that the three policies showed similar performance results in cycle 1, which involves the smallest response messages (in the order of a few Kbytes). This is explained by the fact that, in that cycle, the concurrency overhead generated by parallel invocation is minimal.

3.4. Influence of servers and network characteristics

Even though we could not characterize any of the four UBR servers used in our experiments, neither in terms of their hardware/software configuration nor in terms of their workload, we believe that their individual profiles had little influence on our observed results. This assumption is based on two facts: (i) the four UBR replicas were provided and maintained by industrial-strength IT companies, which means that each of them might have been deployed over a high-end hardware platform with a broadband network connection; and (ii) the replicated UBR service itself lasted only a few years, which probably means that the four servers have never been under high demand. If our assumption holds, the performance variations experienced with the four servers at our two client machines were largely due to load fluctuations in the external network—in addition to differences in the local connection bandwidth available at each client, as we have explained when discussing the results of the parallel invocation policy.

Evidence of the influence of the external network load is provided by analyzing the performance of each server individually. As we have described in Section 3.2.2, IBM and SAP presented similar performance results at both clients, while the performances of Microsoft and NTT varied significantly from client to client. Another source of evidence is the relative unreliability of the probe mechanism used by the HTTPing policy at client UNIFOR, where the slowest server (SAP) was the first to respond to the probe in about 20% of all invocations (see Section 3.3.2).

An analysis of the communication infrastructure connecting the two clients in Brazil to the four servers in their respective continents helps to shed some light into the impact of external network characteristics on our results. Brazil only has direct Internet connections to Europe and North America. The connection to Europe uses four 2 Mbps links to Portugal. The connection to North America is much more robust, and uses several 155 Mbps links to different US backbones (UUNET, Sprint, etc.). All communication with Asia is via either of those two connections, most probably using one of more of the US backbones. The higher capacity of the US links might explain why the two American servers (Microsoft and IBM) offered the best response times, followed by the servers in Asia (NTT) and Europe (SAP), in that order. Also, the fact that IBM and SAP presented similar performance variations at both clients seems to indicate that communication with any of those two servers always involved the same international backbone, regardless of which client was actually the source of communication. On the other hand, the fact that Microsoft and NTT performed differently at each client suggests that communicating with them might have involved different international backbones, depending on which client backbone the communication was initiated.

4. Discussion

The major limitation of our work is certainly the fact that our assessment involved only one replicated service and two different client environments. Even though the experiments were carried out using four publicly available service replicas distributed world-wide, their relative low number, coupled with the fact that the two client machines were both physically located in the same geographic area, also precluded further generalization of our results. Another limitation concerns the fact that the best last and best median policies, which provided the best results overall in our experiments, are strongly dependent on the existence of up-to-date state information about each of the replicated servers. These limitations imply that we must improve the reliability and quality of our experimental work before we can claim that the results observed in our study may generalize to a broader range of service replication scenarios.

With respect to parallel invocation, we need to experiment with a larger variety of client environments, so that we can better understand the relationships between the bandwidth available locally, at the client side, and the concurrency level generated by that policy. This would help us to determine, for example, when using parallel invocation might be recommended or when that policy might be prohibitive for a giving client configuration.

We also need to deepen our investigations on the circumstances under which the probe mechanism used by HTTPing could be affected by external network conditions, so as to improve its effectiveness in predicting service behavior.

Regarding the best median policy, more experiments are necessary to investigate which set of configurations parameters (i.e., the size of the history log and the number of past invocations results used to calculate the median)
would provide the best performance results in a given replication scenario. Concerning the need to have accurate information about the state of each service replica, in our study we have addressed this issue by first invoking the four UBR servers individually, one at a time, at the beginning of each invocation cycle. However, a more general solution would require periodic monitoring of each service replica. This could be done, for instance, using an external server monitor that could be invoked at any time by the underlying server selection mechanism.

5. Related work

There is a growing body of work available in the literature on techniques and tools for dynamic web service selection. In this section, we only discuss those approaches that address specifically the problem of selecting a service amongst a set of replicated or functionally equivalent web services, since they are more closely related to our work.

Costa et al. (2004) describe a web service composition framework that supports dynamic selection of semantically equivalent web services based on a number of quality and cost criteria. The framework offers three basic service selection strategies: best monetary cost, best performance (response time), and best trade-off 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 their claim. In contrast to the work of Costa et al., we have implemented a variety of replica selection policies and empirically evaluated them 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 service selection policy, and that the perceived service response time under that policy may vary significantly, being strongly dependent on the characteristics of the local client environment.

Parallel invocation strategies are also used as part of the web service invocation process. For instance, the authors propose parallel 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. 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 have a strong impact on 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.

Another mechanism for the dynamic web service selection has been proposed by Padovitz et al. (2003). Differently from our work and the works of Costa et al. and Keidl and Kemper, Padovitz et al. focus 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, etc. 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 server state information as they are directly provided by the servers themselves. In our work, in contrast, server selection is based on server performance information as it is perceived at each client machine.

Several web service selection strategies also have been studied in the context of a number of approaches that aim at extending the standard SOC model with explicit QoS support. In general, those approaches introduce novel QoS-based models and/or mechanisms to allow web service providers to publish the QoS properties (e.g., failure rate, response time, reliability, server workload, monetary cost, etc.) of their services (e.g. Zhou et al., 2004; Tian et al., 2004; Yu and Lin, 2005a; Serhani et al., 2005), and also to allow client applications to dynamically collect QoS service information of interest and then select the best service available (amongst a set of functionally equivalent services), based on a variety of user-defined QoS constraints (e.g. Ouzzani and Bouguettaya, 2004; Liu et al., 2004; Tian et al., 2004; Zhuge and Liu, 2004; Hu et al., 2005; Serhani et al., 2005; Yu et al., 2005b; Makris et al., 2006). From a service selection perspective, those solutions have the added benefit that, in principle, they can select services based on any desired QoS criteria (Ran, 2003). This contrasts to our work, in which we select service replicas based on a single QoS attribute (i.e., response time). On the other hand, since those solutions require modifications to the standard SOC model, they also have the important limitation that they cannot be readily applied to most existing...
services. For example, it would not be possible for any of those solutions to select a service-based solely on criteria such as server failure rate or server workload, unless that information is made explicitly available for clients by the target web services’ providers – a feature that is still unsupported by most existing web services development frameworks. This limitation is reflected in the fact the most of the QoS-based service selection strategies proposed thus far have not been validated in any real-world web service replication scenario.

Finally, we call attention to the fact that none of the solutions discussed above addresses the consistency problems typically associated with the invocation of truly replicated services. In particular, if one or more of the provided service operations are likely to affect the replicated service’s internal state, then it is not safe to invoke any of those operations unless there is a replication mechanism in place to prevent clients from accessing inconsistent or out-of-date replicas. The WS-Replication framework (Salas et al., 2006) has recently been proposed specifically to address those problems. To prevent unsafe access to the replicated service, and also to guarantee that all replicas reach a consistent state, WS-Replication uses WS-Multicast, a SOAP-based group communication service, to reliably propagate client requests to the set of service replicas (Salas et al., 2006). Differently from the work on WS-Replication, in our work we do not attempt to address any issue related to replica consistency. However, to prevent replicas from reaching an inconsistent state, we assume that either (1) the requested service operations are idempotent (i.e., they do not have any significant impact on the services’ internal state, as it was the case with the GetServiceDetail operation used in our experiments with the UBR service) or (2) there is a replication mechanism such as WS-Replication in place, that will guarantee replica consistency when non-idempotent operations are invoked.

6. Conclusions

This paper presented our work on the implementation and empirical assessment of five representative server selection policies for accessing replicated web services. These five policies were quantitatively and qualitatively evaluated, at two distinct client settings, in terms of their observed performance results when used to access a world-wide replicated web service.

Based on our assessment’s results, we have formulated a set of guidelines to help web services developers and users in identifying a server selection policy that is more appropriate for their replication scenario at hand. These results and guidelines are summarized below:

- Server selection policies that take into account past invocation results to drive their future server selection decisions (such as best last and best median) tend to offer the best performance overall, independently of the characteristics of the replicated servers or the local client environments. Therefore, those policies may be the ideal choice when little information is known about the replication scenario at hand. On the other hand, this type of policy might not be recommended if the replicated servers are only invoked sporadically, or their response times are subject to frequent and abrupt changes.
- The performance provided by server selection policies that are highly dynamic in nature (such as parallel invocation and HTTPing) tends to be strongly influenced by the characteristics of the local client environment, such as the available bandwidth, or variations in the external network load. Therefore, developers must carefully consider those factors before committing to any policy of this type. For example, parallel invocation might be recommended for scenarios where there is enough local bandwidth to handle the generated concurrent traffic; the number of service replicas is low; and load balance is not a major system concern. HTTPing, in turn, might be recommended as a viable alternative to parallel invocation when there are many service replicas available or the servers’ workload is high. However, the benefits of HTTPing might be compromised if load variations in the external network are likely to affect the reliability of its probe mechanism.
- Server selection policies that treat each service replica equally (such as random selection) should be used only when the remote servers and the external network are guaranteed to provide different client applications with similar performance levels. Otherwise, this type of policy is likely to have a strong negative impact on clients, since it tends to select low performance servers as often as the most responsive ones.

As we have discussed in Section 4, our contributions are still limited in several ways. In addition to the research lines suggested therein, we envision a number of other interesting directions upon which our work could be improved or extended. First and foremost, we plan to conduct more experiments involving new services, possibly replicated over a larger number of geographic locations, and new client settings, in an attempt to investigate the extent to which the results reported in this paper could be generalized.

Another research line worth pursuing is to experiment with novel server selection strategies, possibly by improving upon some of the policies already implemented. For example, we plan to investigate how the best median and parallel invocation policies could be combined, so that only those servers with the best past performance results would be invoked in parallel. Such strategy would certainly reduce the concurrency overhead that is associated with parallel invocation, and would be specially attractive to replication scenarios in which the target web service is replicated over a large number of servers.

With respect to our implementation strategy, one promising research line would be to re-implement the server selection mechanism in the form of a proxy service. This
would allow extending its server selection facilities to benefit other non-Java-based client applications, without the need to change their source code or their underlying software development framework. Another advantage of this approach is the opportunity for history-based server selection policies (such as best last and best median) to share information on past invocation results originated from different client applications. This is not possible with the current version of our implementation, given that the AXIS-based server selection mechanism can only be deployed in a per application basis.

To conclude, we stress that we are aware that by no means performance is the only QoS attribute that should be considered at the client-side when selecting amongst a set of replicated or functionally equivalent web services. Factors such as security, reliability, transaction support, replica consistency, and even monetary cost, may all play important roles during the service selection process. However, in view of current practice in service-oriented development, and the fact that most existing web service technologies still lack adequate QoS support, performance is likely to remain the dominant selection criteria, at least in the foreseeable future. Therefore, an empirical evaluation of the performance impact imposed by a variety of representative service selection strategies, carried out in a real-world Internet setting, as reported in this paper, can be an important contribution to foster further experimental work in this relevant research field.

Acknowledgements

We are grateful to the anonymous reviewers, whose comments and suggestions significantly improved the paper. We also thank André Coelho for his help with an early draft of the manuscript.

References


Proceedings of the USENIX Annual Technical Conference. Anaheim, California, USA.

Nabor C. Mendonça is a titular professor at the Center of Technological Sciences, University of Fortaleza, Brazil. Formerly, he was a visiting researcher at the Department of Computer Science, Federal University of Ceará, Brazil. His research interests include distributed systems, service-oriented development, aspect-oriented programming, and software maintenance and reengineering. He holds a Ph.D. degree in Computing from Imperial College London, UK. He is a member of the IEEE Computer Society and the Brazilian Computer Society.

José Airton F. Silva is a system analyst at Banco do Nordeste, Brazil. His research interests include distributed systems and service-oriented development. He holds an M.Sc. degree in Applied Informatics from University of Fortaleza, Brazil.

Ricardo O. Anido is an associate professor at the Institute of Computing, State University of Campinas, Brazil. From 1994–1996 he was a visiting researcher at the Institut National Des Télécommunications, France. His research interests include distributed systems, fault-tolerant systems, and distributed algorithms. He holds a Ph.D. degree in Computing from Imperial College London, UK.