Robust Web Services Provisioning Through On-Demand Replication

Quan Z. Sheng[1], Zakaria Maamar[2], Jian Yu[1], and Anne H.H. Ngu[3]

[1] School of Computer Science, The University of Adelaide, SA 5005, Australia
{qsheng,jyu}@cs.adelaide.edu.au
[2] College of Information Technology, Zayed University, Dubai, UAE
zakaria.maamar@zu.ac.ae
[3] Department of Computer Science, Texas State University, San Marcos, TX, USA
angu@txstate.edu

Abstract. Significant difference exists between things that work and things that work well. Availability along with reliability would make Web services the default technology of choice in developing many mission-critical electronic-business applications. Unfortunately, guaranteeing a Web service availability is still a challenge due to the unpredictable number of invocation requests a Web service (e.g., Google Maps) has to handle at a time, as well as the dynamic nature of the Internet (e.g., network disconnections). In addition, the heterogeneity, mobility and distributed nature of Web services makes traditional dependability and availability approaches inappropriate for Web services. In this paper, we describe the design of an on-demand replication approach for robust Web service provisioning. This approach dynamically deploys Web services at appropriate idle hosts, reducing the unavailability of Web services during peak demand for limited computing resources. In addition this approach promotes the decoupling of Web service providers and Web service host providers, which supports a more flexible replication model.

Key words: Web service, replication, service host, matchmaking

1 Introduction

The emerging Service-Oriented Computing (SOC) paradigm highlighted by Web services technology promises bringing better interoperability and flexibility to business applications. In order to compete globally, more and more enterprises turn nowadays towards Web services to achieve better visibility and accessibility of their core business competencies (e.g., Amazon Web services[1]). Unfortunately, over the past few years, efforts put into SOC were mainly concentrated on making things (e.g., Web services integration) work, but barely on making these things work well [3, 8]. This posed major challenges to enterprises wishing to embrace Web services as a development technology for their IT critical applications.

One such challenge, yet still being largely overlooked, is the *availability* (readiness of correct service) and *reliability* (continuity of correct service) of Web services [3, 4, 8, 10]. Given the large number of requests that Web services handle

[1] http://aws.amazon.com/.

(e.g., Google was searched 4.4 billion times in the US alone in October 2007[2]), computing hosts upon which these Web services operate can easily turn out to be inadequate for guaranteeing low response-times and unavailability in case of heavy loads. However, enterprises cannot afford interrupting their activities, even for short periods of time. For instance, in April 2002, eBay suffered a 22-hour server-outage affecting most of its online auctions, costing five million US dollars lost in revenue and over five billion dollars in market capitalization.

A recent emerging trend for solving the high-availability issue of Web services is centered around *replication* [4, 10]. The underlying idea is to spread replicas over various computing hosts and direct invocation requests to appropriate hosts (e.g., with lower workload). The aim of the work reported in this paper is to enhance the fundamental understanding of robust Web services provisioning and to develop a novel approach by advancing the current state of art in this direction. Our main contributions are summarized as follows:

- Web services are deployed on-demand by targeting appropriate idle computing hosts. This reduces the unavailability risk and achieves better load-balancing. A core architectural design in this deployment is the clear separation between *Web service* providers and *Web service host* providers.
- Dynamic Web services replication model that enables proactive deployment of replicas so that a certain desired availability level is maintained. This model helps determine how many replicas are needed, when and where they should be created and deployed. The deployment of replicas remains transparent to clients so that they are unnecessarily aware of the underlying replication.
- Mechanisms for Web service hosts matchmaking and selection. A *matchmaking* process and a *multi-criteria utility* function are introduced to dynamically select appropriate computing hosts based on Web services' requirements.

The remainder of the paper is organized as follows. Section 2 overviews the basic concepts that underpin the robust Web service provisioning approach and Section 3 introduces the system design. Section 4 describes the solution on dynamic replication of Web services. Section 5 introduces our algorithm for service host matchmaking. Implementation and some experimental results are documented in Section 6. Finally, Section 7 concludes the paper.

2 Preliminaries

This section introduces some basic concepts upon which our robust Web services provisioning approach is built namely, Web services anatomy, Web service hosting systems, and service code mobility.

2.1 Anatomy of Web Services

Interacting with Web services relies on exchanging SOAP messages. These messages contain various details such as input/output data format that are specified and included in WSDL documents. Figure 1 depicts a typical interaction-session

[2] http://www.buzzmachine.com/2007/12/29/google-is-god/.

with a Web service. Initially, a provider informs potential clients of the address (i.e., endpoint) that its Web service uses to receive SOAP messages. This address is posted on a registry such as UDDI. Upon reception, a message is converted by the Web service's processing logic into a form that is appropriate for the computing system upon which this Web service runs.

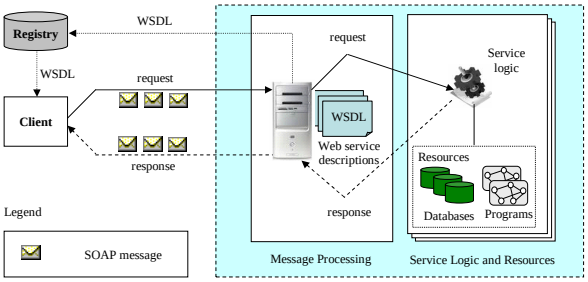


Fig. 1. Structure of a typical interaction-session with a Web service

The separation of concerns on business logic and message handling, which is actually a fundamental property of the concept of Web service, offers several advantages: i) free evolution of Web services (e.g., performance improvement) without affecting binding mechanisms of service consumers, and ii) free decoupling of Web service's endpoints (i.e., locations where SOAP messages are sent) from Web service invocation (i.e., locations where the Web service is executed).

2.2 Web Services and Web Service Hosts

Inspired by the Web site hosting trend where business (and personal) Web pages are stored in one or multiple hosting service providers (e.g., `HostMonster`[3]), we advocate the separation of *Web services* providers and *host* providers.

A Web service implements the functionalities of an application, developed by a particular provider. This provider is also responsible for advertising the Web service to potential customers and installing it on appropriate hosts, either local or distant. A Web service can be concurrently made available through replicas on distinct service hosts. To deal with service hosts variety (e.g., Windows, Linux and Solaris), a Web service provider needs to maintain ready-to-deploy versions of this Web service according to the hardware and software requirements of these hosts. A deployable version is typically a Web service bundle that contains all what is necessary to deploy the service in a single file (e.g., war or jar file).

A host provider has the authority to control the computing resources (e.g., a set of servers installed Apache Axis) it manages for the sake of accommodating the execution needs of Web services. It accepts SOAP messages sent from a Web service provider, routes these messages to the host where the corresponding Web service is located, and returns Web service results (if any), in another SOAP message, to the Web service provider.

[3] http://www.hostmonster.com/.

2.3 Stationary and Mobile Web Services

It is accepted that code mobility is suitable for load balancing, performance optimization, and disconnection handling [5]. Code mobility would significantly contribute to secure prompt responses; applications running on heavy loaded or unavailable hosts can be dynamically migrated to other better hosts. Examples of mobile code technologies include Aglets[4], Java, Sumatra [1], and μCode[5].

Injecting code mobility into Web services sheds the light on *stationary* and *mobile* Web services. Stationary Web services are location-dependent; they can not move because for various reasons such as resource availability (e.g., a specific database) on a dedicated host. On top of behaving like their counterparts in terms of accepting invocation requests and taking part in composition scenarios, mobile Web services are i) location-independent, ii) enhanced with migration capabilities that make them move to distant hosts for execution, and iii) stateless. As a result, a mobile Web service can be executed on any arbitrary host as long as this host accommodates this mobile Web service's execution needs.

Mobile Web services have already attracted the attention of the research community as the large number of references witness [2, 6, 7]. A simple way is to pack the bundle of a mobile Web service into a single file (e.g., war file) that can be dynamically deployed at appropriate service hosts. In [2], mobile Web services are considered as synthesis of Web services and mobile agents. In [7], Liu and Lewis develop an XML-based mobile code language called X# and presents an approach for enabling Web services containers to accept and run mobile codes. It should be noted that proposing techniques for the development of mobile Web services is not the focus of this paper. Instead, we adopt some of the strategies used in mobile Web services in our approach for dynamic replication of Web services that can move around.

3 System Design

Central to our approach for making Web services provisioning robust are the following components (Figure 2): *Web service provider*, *Web service manager*, pool of *Web service hosts*, and *matchmaker*.

The Web service provider develops Web services and stores their deployable versions in a Web service code-repository. Each Web service provider is associated with a Web service manager that consists of four components, namely *service controller*, *service replicator*, *service register*, and *service monitor*, which collaborate together to achieve and sustain Web services high-availability.

The service monitor oversees the execution status of a Web service (e.g., exceptions). The service controller interacts with the service monitor and coordinates the execution of the corresponding Web service. If the value of a monitored item (e.g., CPU usage) is beyond a critical point—which can be set by the service provider—and the Web service is movable/mobile [2], the controller invites the service replicator to move the Web service to a selected Web service host

[4] http://www.trl.ibm.com/aglets/.
[5] http://mucode.sourceforge.net/.

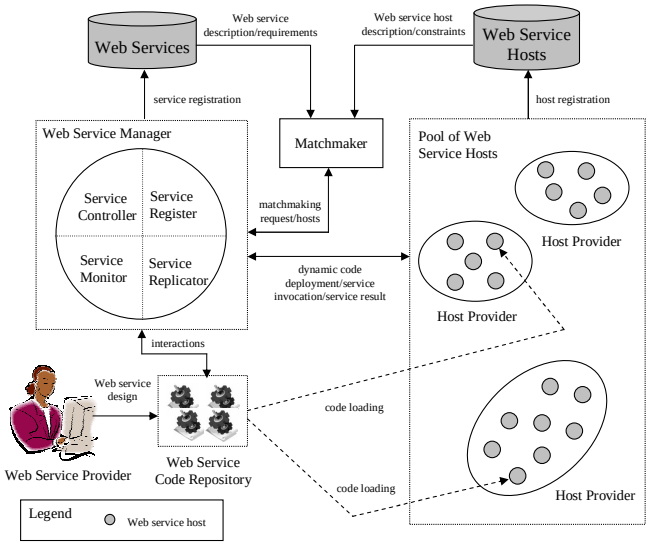


Fig. 2. System architecture

for execution. If such a Web service host does not exist or the Web service is stationary, the controller triggers an exception handling policy (e.g., forward the Web service invocation to a substitute peer), if such a policy has been specified by the service provider. Different solutions to Web services substitution are reported in the literature such as [9].

The service replicator is a light weight scheduler that helps the controller execute the associated Web service on other computing resources when necessary. The replicator decides i) the number of replicas needed for a desired availability degree, and ii) when and where the replicas are deployed. Web service codes are transferred to hosts and invocation requests are dynamically routed to these Web services' new locations. Its underlying technique is what we call *dynamic service replication* and more details are reported in Section 4.

Each host provider controls a cluster of Web service hosts. These hosts have monitoring modules (not shown in the figure for the sake of simplicity) that oversee resource consumption in terms of CPU, memory, etc. Web services and Web service hosts can register themselves using public registries like UDDI. To this end, an appropriate tModel[6] (e.g., `ServiceHost`) to describe hosts is required so that Web services can locate them. Service host selection is another important factor to the success of our approach. The selection of service hosts is based on a matchmaking mechanism that is implemented in the matchmaker. More details on host matchmaking are given in Section 5.

[6] In UDDI, a tModel provides a semantic classification of the functionality of a service or a host, together with a formal description of its interfaces.

4 Dynamic Web Services Replication

Replication is well-known to improve system availability. In this section, we introduce a mechanism that permits to proactively create Web service replicas so that the availability requirement is sustained[7]. This mechanism relies on a *replication decision model* that helps a Web service determine how many replicas are needed, and when and where they should be created and deployed.

4.1 Replication Decision Model

In replication there is always a trade-off between availability and cost. On the one hand, more hosts accepting a Web service means better and higher availability. On the other hand, more replicas at various hosts means higher overhead (e.g., resource consumption such as bandwidth and storage space). It is, therefore, essential to have a replication decision model, which helps the service replicator determine i) the optimal number of the replicas in order to meet a Web service's availability requirement, and ii) time and locations for deploying the new replicas.

Calculating the Number of Replicas. Given the failure probability of each Web service host p, total number of replicas $\mathcal{N}_t$ of a Web service s, and availability threshold $\mathcal{A}$ of s, the following formula must be satisfied,

$$\mathcal{A}(s) <= 1 - p^{\mathcal{N}_t(s)} \tag{1}$$

where $p^{\mathcal{N}_t(s)}$ represents the probability of all replicas of Web service s being unavailable and 1-$p^{\mathcal{N}_t(s)}$ represents the probability of at least one replica of s being available. The meanings of the notations are given in Table 1. From Formula 1, we can easily get

$$\mathcal{N}_t(s) >= \log_p^{1-\mathcal{A}(s)} \tag{2}$$

to calculate the right number of replicas for a certain availability threshold. For example, assume that the failure probability of service hosts p is 10% (low). To satisfy the availability threshold of 99.99% of a Web service, 4 replicas are recommended. If p is 50% (medium), the number of replicas is now 14. If p becomes 90% (high), this number increases to 88.

Notation	Meaning
$\mathcal{A}(s)$	Required availability threshold of Web service s
p	Failure probability of each service host
$\mathcal{N}_t(s)$	Total number of replicas needed for the Web service s
$\mathcal{N}_e(s)$	Total number of replicas currently existed for the Web service s

Table 1. Notations and their meanings

[7] In the remainder, we assume that the Web service is mobile when Web service replication is referred to.

Deploying Replicas. Once the number of required replicas (i.e., $\mathcal{N}_t$) of a Web service s is known, the service replicator of s might decide to deploy additional replicas of s on top of the existing ones (i.e., $\mathcal{N}_e$). The service replicator takes no action if $\mathcal{N}_t(s)$ is less than or equal to $\mathcal{N}_e(s)$. Otherwise, the replicator has to deploy $\mathcal{N}_t(s) - \mathcal{N}_e(s)$ replicas of s to remote hosts. This typically involves the following tasks: i) selecting suitable service hosts for the replicas (Section 5), and ii) deploying replicas to the selected service hosts. A simple way for obtaining the value of $\mathcal{N}_e$ is to count the Web service hosts that accepted replicas of the Web service, which is maintained in the *routing table* to be detailed in Section 4.2.

Due to dynamic nature of the Internet (e.g., retraction of a service host without prior notice), an interesting problem is that when a Web service should check whether it needs to deploy more replicas in order to meet the required availability level. We propose that the service replicator should periodically compare $\mathcal{N}_t(s)$ with $\mathcal{N}_e(s)$. The periodicity of the checks depends on the availability level of the Web service. In case of critical availability, a shorter frequency is applied, which means more frequent checks. Otherwise, a longer frequency is applied. The service replicator can also consider the checking history. For example, if there are no actions needed during the last few (e.g., 3) checks, the replicator can increase the check interval. Contrarily, if new replicas were needed in the last few checks, the replicator would decrease the check interval.

4.2 Deployment Transparency

Transparency is an important issue from consumers' point of view. They should by no means be aware of the existence of replicas. In our design, a provider registers a Web service in a UDDI repository by making the service controller (Figure 2) act as this Web service's endpoint. It is the service controller's responsibility to receive invocation messages and forward them to the appropriate service hosts for actual service invocation. This is achieved with the help of a *routing table* that the service replicator maintains. This table keeps the access information on the deployed copies by simply recording a pair of <`Host-ID`, `EndPoint`> where `Host-ID` is the identifier of the service host upon which the Web service is deployed and `EndPoint` is the access point of the deployed replica.

It should be noted that a replica deployed on a host may not be working due to reasons that could be related to end of contract or giving room to higher priority Web services. For the sake of simplicity, we assume that the routing table keeps updated information. For example, if a deployed Web service is removed from a service host, its record will be removed from the routing table as well. When the need for service invocation on a service host rises, the service controller consults the routing table and forwards (routes) the service invocation message to the other host(s) for processing. Strategies that the controller can apply include: i) selecting the best service host from the routing table for invocation, and ii) selecting all the service hosts for invocation and returning the first response to the customer.

Figure 3 illustrates the high level interactions among the components during Web service invocation. When a client invokes a Web service (step 1), the ser-

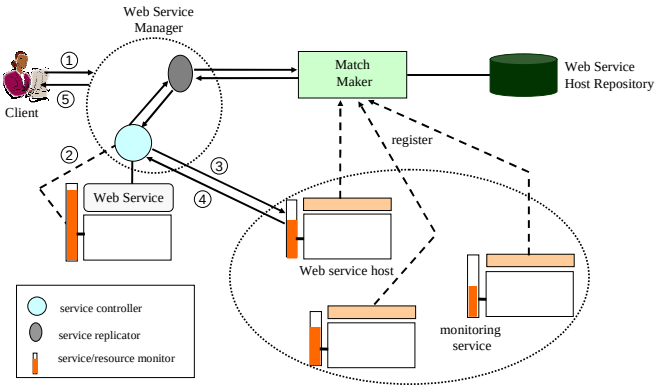


Fig. 3. Interactions of system components for Web service invocation

vice controller[8] contacts the service monitor about the status of the local service host (step 2). If the host is overloaded (e.g., CPU usage of the host is higher than 90%), the controller picks up a service host, by consulting with the service replicator and the matchmaker and forwards the invocation message to the endpoint of the corresponding replica (step 3). The service result is then returned to the controller (step 4). Finally, the Web service manager returns the result to the client (step 5).

5 Host Matchmaking and Selection

The approach we have presented relies on a *matchmaking* mechanism to locate and select appropriate Web service hosts. The basic idea of host matchmaking is summarized as follows: service replicators and host providers advertise requirements and characteristics of Web services and service hosts; a designated module (i.e., *matchmaker*) matches the advertisements in a manner that meets the requirements and characteristics specified in the respective advertisements.

The descriptions of Web services and resources consist of two parts: *attributes* and *constraints*. The attributes for a Web service includes characteristics such as service location, mobility, and input/output parameters. The attributes for a host include properties such as CPU usage, free memory, and price. The constraint part includes limitations defined by the providers of Web services and hosts. For example, a host provider may specify that it will not service any request coming from company A due to regular payment delays. Similarly, the provider of a (mobile) Web service may specify that only hosts with more than 200K bytes of free disk space and at least 128K bytes of free memory are eligible to host this Web service. Service hosts can be described using W3C's RDF (Resource Description Framework)[9], while Web services can be described

[8] For the sake of simplicity, other components in the Web service manager are excluded from Figure 3.

[9] http://www.w3.org/TR/REC-rdf-syntax.

using WSDL (Web Service Description Language)[10]. We do not give detailed description of RDF and WSDL due to space limitations.

Matchmaking Interactions. Matchmaking is defined as a process that requires a service description as input and returns a set of potential hosts. A host is matched with a service if both the requirement expressions of the service and the constraints of the host are evaluated to true.

Several steps involve in the interactions of the matchmaking process. Service replicators and host providers develop descriptions for their requirements and attributes and send them to the matchmaker (step 1). The matchmaker then, invokes a *matchmaking algorithm* by which matches are identified (step 2). The invocation includes finding service-host description pairs that satisfy the constraints and requirements of hosts and services. We will detail this step later. After the matching step, the matchmaker notifies the service replicator and the host providers (step 3). The service replicator and the host provider(s) then contact each other and establish a working relationship (step 4). It should be noted that a matched host of a service does not mean that the host is allocated to the Web service. Rather, the matching is a mutual introduction between Web services and hosts and the real working relationship can be consequently built after the communication between the two parts.

Expression Evaluation. Expression evaluation plays an important role in the matchmaking process. To evaluate a constraint expression on a service description, the attribute of the expression is replaced with the value of the corresponding attribute of the host. If the corresponding attribute does not exist in the host description, the attribute of the expression is replaced with the constant `undefined`. In our matchmaking algorithm, expressions containing `undefined` are eventually evaluated to *false*. The constraints of the host descriptions have the similar evaluation process. When receiving a request from a service replicator, the matchmaker takes the Web service description, evaluates all the hosts advertised in the host repository using the matchmaking algorithm described above, and returns a set of matched hosts to the service replicator.

For example, assume that a Web service needs at least 128K bytes free memory to run (i.e., `memoryfree`>=128K). The matchmaker scans the description of a host for the attribute `memoryfree`. The value of the attribute (e.g., 512000K bytes) is used to replace the attribute in the expression (i.e., 512000 >= 128), which is in turn evaluated to *true* by the matchmaker.

Hosts Selection. For a specific Web service, there could be multiple potential hosts matched for executing the Web service. The service provider, therefore, should be able to choose the best host (or top N best hosts) that satisfies its particular needs from the matched hosts. To specify preferences over hosts of a particular Web service, we exploit a *multi-criteria utility function*,

$$\mathcal{U}(h) = \sum_{i \in \mathcal{SA}} w_i \cdot \mathcal{S}core_i(h) \tag{3}$$

[10] http://www.w3.org/TR/wsdl.

where i) h is a host, ii) $\mathcal{S}core_i(h)$ is an attribute scoring function, iii) $\mathcal{SA}$ is the set of selection attributes, and iv) w_i is the weight assigned to attribute i. The scoring service computes the weighted sum of criteria scores using the weight property of each selection criterion. It selects the host that produces the higher overall score according to the multi-criteria utility function. Several criteria—such as *price*, *availability*, *reliability*, and *reputation*—can be used in the function. Due to the space constraint, the criteria calculation is not described in this paper. Interested readers are referred to [11] for details.

6 Implementation and Experimentation

6.1 Implementation

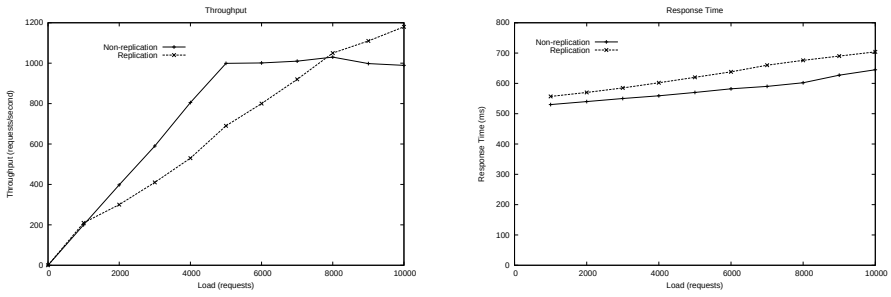
This section describes the system implementation with focus on the matchmaker and the Web service manager (Figure 2).

The Matchmaker. Two repositories are included in the system, namely *Web service repository* and *Web service host repository*. Both repositories are implemented as UDDI registries. Each host is represented as an RDF document. A separate tModel of type `hostSpec` is created per Web service host, including a tModel key, a name (i.e., host name), an optional description, and a URL that points to the location of the host description document. WSDL is used to specify Web services. Since WSDL focuses on how to invoke a Web service, some of the attributes (e.g., stationary or mobile service) proposed in our approach are not supported. To overcome this limitation, such attributes are specified as tModels. The keys of these tModels are included in the `categoryBag` of the tModel of a Web service so that the Web service knows the descriptions of these attributes.

The matchmaker is implemented using the UDDI Java API (UDDI4J) provided by IBM Web Services Toolkit 2.4 (WSTK). This matchmaker provides two kinds of interfaces for both repositories, including an *advertise interface* and a *search interface*. The former interface is used to publish Web services and service hosts, while the latter interface is used to search service hosts using the approach presented in this paper.

The Web Service Manager. The functionalities of the service controller are realized by a pre-built Java class, called `Controller`, which provides an operation called `coordinate` to receive invocation messages, manage service instances (e.g., initializing an instance in a remote host with a replica of the Web service), trace service invocations, and trigger service replication actions.

The controller relies on the service monitor and the service replicator to make intelligent decision on when and where to perform the replication. In particular, the service replicator is the module that maintains a desired availability degree of a Web service. It provides a specific method called `replication` that can be used to perform replica number calculation and replica deployment. In addition, the service replicator maintains a routing table that contains the information on the replicas of a Web service (e.g., endpoints of the replicas). Currently, the routing table is simply implemented as an XML document.



(a) (b)

Fig. 4. Performance study: (a) throughput, (b) response time

6.2 Experimentation

In this section, we present some preliminary experimental results with emphasis on evaluating the performance overhead introduced by our approach. We implemented a simple Web service that takes a string *str* as input and outputs another string which simply joins `Hello` and *str* together. The service is deployed using Apache Axis. Four machines were used in the experiments. One machine was dedicated to the original Web service (including the Web service manager) and three were used to host replicas of this Web service. All machines have the same configuration of Pentium III 933MHz and 512Mb RAM, and are connected to a LAN through 100Mbits/sec Ethernet cards. Each machine runs Apache Axis 1.2 with Jakarta Tomcat 5.5, Debian Linux, and the Java 2 Standard Edition V1.4.2.

To evaluate the performance overhead of our approach, we considered two cases: i) invoking the string-joining Web service directly (i.e., non-replication case where service requests are sent to the Web service), and ii) invoking the string-joining Web service using our approach (i.e., replication case where service requests were sent to the service controller of the Web service). We simulated 10 clients, in another machine, which submitted invocation requests simultaneously to both cases respectively for service execution. The number of request messages ranges from 1,000 to 10,000.

Figure 4 (a) gives the average throughput of the two cases. We can see that the throughput of the non-replication case reaches its maximum around 5,000 requests. After that, the throughput remains steady and does not increase anymore. The throughput of the replication case, on the other hand, keeps increasing even when the number of requests reaches 10,000. Figure 4 (b) shows the average response time. We can see that there is a relatively small overhead in the replication case. For example, when the number of request messages reaches 6,000, the average response time of non-replication case is 582ms, whereas for the replication case, it is 638ms. This overhead is reasonable due to the additional method calls that are inherent to our Web service manager implementation.

7 Conclusion

With rapid adoption of Service-Oriented Computing (SOC), more and more organizations are deploying their business competencies as Web services over the

Internet. In these scenarios, service utility is often determined by its availability rather than the traditional metric of raw performance. Unfortunately, guaranteeing a Web service availability is still a challenge due to the unpredictable number of invocation requests a Web service has to handle at a time, as well as the dynamic nature of the Web. In this paper, we presented an approach that exploits service replication to achieve robust Web services provisioning. The main features of this approach are: i) a dynamic Web service replication model to proactively deploy replicas of a Web service for a desired availability level, and ii) mechanisms for Web service hosts matchmaking and selection. The preliminary results show that the cost of high availability (i.e., overheads) is acceptable. The results also show that our approach achieves a better throughput.

Our ongoing work includes further performance study on the on-demand replication strategy (e.g., in a WAN environment) of the proposed techniques. This includes simulating more complex composite services with some component services which are location dependence. It also requires more extensive study on composite services which have services that can be executed in parallel. In addition, we plan to look into leveraging the dynamic information of service hosts (e.g., amount of free memory) and Web services (e.g., quality of service) for the matching and selecting of the best host.

References

1. A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Resource-Aware Mobile Programs. *Mobile Object Systems: Towards the Programmable Internet*, pages 111–130, 1997.
2. M. Adacal and A. B. Bener. Mobile Web Services: A New Agent-Based Framework. *IEEE Internet Computing*, 10(3):58–65, May-June 2006.
3. K. Birman, R. van Renesse, and W. Vogels. Adding High Availability and Autonomic Behavior to Web Services. In *Proc. of the 26th Intl. Conf. on Software Engineering (ICSE'04)*, Scotland, UK, May 2004.
4. P. Chan, M. Lyu, and M. Malek. Reliable Web Services: Methodology, Experiment and Modeling. In *Proc. of IEEE Intl. Conf. on Web Services (ICWS'07)*, Utah, USA, July 2007.
5. A. Fuggetta, G. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, 24(5):342–361, May 1998.
6. F. Hirsch, J. Kemp, and J. Ilkka. *Mobile Web Services: Architecture and Implementation*. John Wiley & Sons, 2006.
7. P. Liu and M. Lewis. Mobile Code Enabled Web Services. In *Proc. of IEEE Intl. Conf. on Web Services (ICWS'05)*, Orlando FL, USA, July 2005.
8. M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *Computer*, 40(11):38–45, 2007.
9. J. Ribeiro, G. do Carmo, M. Valente, and N. Mendonça. Smart Proxies for Accessing Replicated Web Services. *IEEE Distributed Systems Online*, 8(12), 2007.
10. J. Salas, F. Pérez-Sorrosal, M. Patiño-Martínez, and R. Jiménez-Peris. WS-Replication: A Framework for Highly Available Web Services. In *Proc. of the 15th Intl. World Wide Web Conf. (WWW'06)*, Edinburgh, Scotland, May 2006.
11. L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality Driven Web Services Composition. In *Proc. of the 12th Intl. World Wide Web Conf. (WWW'03)*, Budapest, Hungary, May 2003.