An Architecture for Highly Available and Dynamically Upgradeable Web Services

Nearchos Paspallis and George A. Papadopoulos

Department of Computer Science, University of Cyprus
75 Kallipoleos Street, P.O. Box 20537, CY-1678, Nicosia, Cyprus
{nearchos, george}@cs.ucy.ac.cy

Abstract. Developing distributed application architectures characterized by high availability has always been a challenging and important task both for the academic and the industrial communities. Additionally, the related requirement for dynamic upgradeability is usually examined within the same context as it also aims for high availability. Although a number of architectures and techniques have been proposed and developed for improving the availability and upgradeability of traditional distributed systems, not many of them are directly applicable to Web service-based architectures. Recently, Web services have become the most popular paradigm for business-to-business and enterprise application integration architectures, which makes their availability increasingly important. This paper builds on existing high availability and dynamic upgradeability techniques which can be applied to Web service-based systems. Based on them it describes an architecture which enables high availability and dynamic upgradeability both for newly developed and for prefabricated Web services.

Keywords: Dependable systems, Service engineering.

1 Introduction

Web services are the technology-of-choice for interoperability within non-homogeneous systems. They are briefly defined as *"self-contained, modular applications that have open, Internet-oriented, standards-based interfaces"* [3]. Although many expect that Web services will change the way enterprises interoperate with each other in the long-term, they have already proven themselves very useful in solving many of the interoperability problems that have troubled application integration efforts.

Today, one can observe more and more enterprises depending on their Web accessible services to a continuously increasing degree. Some of these services are critical and the enterprises invest a lot of effort and resources in maintaining them as highly available as possible. Examples of Web services requiring high levels of availability include medical, stock market, and airline ticket reservation applications.

While a number of techniques target at maintaining the high availability of a service in the exceptional case of a fault, additional effort is also required to maintain

the accessibility of the service when it is undergoing a scheduled update. This is why systems designed for high availability usually provide mechanisms enabling dynamic (also known as live) software upgrades as well. Typical reasons for upgrading software include bug fixes, functionality enrichment and performance enhancements. Naturally, highly available services are expected to be upgradeable in a safe and consistent manner.

This paper studies techniques for achieving high availability of services offered over the Web and also techniques that allow dynamic upgrades of those services. Although some of these approaches could be applied to general distributed systems, this work concentrates on Web services and their underlying technologies (e.g. SOAP, WSDL, UDDI, etc).

In the next section, we examine the Web services technology from the availability and upgradeability point of view. Then, existing techniques for high availability and dynamic upgradeability are presented in section 3. Following that, a methodology for improving on the availability and upgradeability of Web services is proposed in section 4. The proposed method involves automatic generation of stubs and skeletons using the WSDL document. Furthermore the same section argues on the advantages and the disadvantages of this approach and, finally, section 5 summarizes the conclusions and points to future work.

2 Web services

Web services can be described as applications accessible over the Web [3]. More accurately the World Wide Web Consortium (W3C) describes them as: "…*software applications identified by URIs, whose interfaces and bindings are capable of being defined, described and discovered as XML artifacts. Web services support direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols*."

The promise of Web services is to serve as the foundation for a new generation of business-to-business (B2B) and enterprise application integration (EAI) architectures. Because Web services are both language and platform neutral, it is a common practice for enterprises to expose a selected subset of their functionality (of newly developed but also of legacy information systems) as Web services. This ability greatly contributes to the popularity of Web services in both B2B and EAI scenarios.

2.1 Highly available web services

In this paper the availability refers to a measure of the fault tolerance of a Web service. Consequently, high availability is defined as a goal that we try to achieve by employing a number of methods and techniques. Literally, availability is the percentage of time a Web service is available and functioning within its operational requirements. Obviously, to achieve high availability a service needs to maximize its uptime and minimize its downtime.

High availability is important for many enterprises because their system responsiveness is directly related to their customer satisfaction and, consequently, to

their operations turnover. Also for many enterprises which are highly depended on Web services, their downtime is usually proportional to significant revenue losses. In many other applications such as medical information systems, high availability is inherently critical and extremely important.

The availability can be affected by factors such as connectivity (i.e. network availability) and server failures (i.e. hardware and software faults). Thus any solution aiming to provide reasonable protection against failures should cope with both factors.

Based on their definition, Web services need to be discoverable and also facilitate interactions with other systems by continuously allowing binding and interaction. Therefore, to ensure that Web services maintain high availability, their discovery and binding functionality need to be enhanced with appropriate mechanisms. The discovery of Web services is generally performed using service directories based on the UDDI standard, which are Web services themselves. Thus, improving on the availability of general services offered over the web, consequently benefits the discovery of Web services as well.

2.2 Dynamically upgradeable web services

With dynamic upgrades, we refer to the replacement of software components at runtime, with minimal (preferably zero) service interruption.

In the client-server paradigm, the upgrade can take place at either of the two sides, or at both. In most cases, client-side upgrades are more straightforward compared to server-side upgrades because they can take place in a controlled manner (i.e. the client can be instructed to suspend or drop any connections to the server or even completely shut an application down if necessary.) This eases the task of replacing some components or the whole application.

Contrary to this, upgrading server side components is significantly more challenging. Because no pre-determined downtime is known, clients initiate transactions with the service in an arbitrary way. In [9] Kramer *et al.* introduced the notion of quiescence, i.e. a period within which the component can be safely upgraded (e.g. replaced.) A component is said to be quiescent when the component itself and all components linked to it are in a passive state, i.e. seize initiating but continue serving transactions.

Dynamic upgrades are required for a number of reasons. These are classified as corrective, perfective and adaptive [11]. Corrective upgrades are used for fixing bugs (e.g. discovered after deploying the service). Perfective upgrades are used to enhance the product functionality and performance and, finally, adaptive changes are needed for adjusting services to a changing environment.

Dynamic upgradeability is important because it enables continuous service operation in the event of scheduled upgrades. Thus mechanisms for dynamic upgradeability are important (and quite often required) supplements to high availability architectures.

3 Related work

This section reviews existing high availability and dynamic upgradeability techniques with emphasis on those targeting server-side faults. While complex upgrade mechanisms have been proposed, a common approach employed by high availability architectures includes temporary redirection of the traffic before upgrading the server, and then redirection of the traffic back to the original server once that is completed.

It is worthwhile mentioning that different techniques operate at different layers of the system architecture. At the lowest layer some techniques use hardware replication, while at the highest layer other techniques simply embed the mechanisms required for high availability into the applications themselves.

There are a number of criteria that can be considered while evaluating architectures. Here, we concentrate on the transparency of the investigated techniques and their applicability to prefabricated Web services. By transparent we refer to those techniques which can be applied without requiring any major changes to existing infrastructures, i.e. those that can be directly applied to existing Web services.

3.1 Existing techniques

In [4] Birman *et al.* discuss methods for adding high availability to Web services. In addition, they also discuss how to enable autonomic behavior i.e. how to enable servers to automatically discover and configure themselves and then operate securely and reliably in an automated manner.

Their work uses extensions to the Web services model which aim to support standard services for monitoring the health of the system, self-diagnosis of faults, self-repair of applications and event reporting. The solution builds on existing technologies, such as WS-Transactions [6] and WS-Reliability [8] but it eliminates their need to save data to persistent memory or wait for failed components to restart.

This solution can act as a router component of a Web service platform, making it suitable for providing transparent high availability to existing applications. However this approach does not define explicit methods to enable dynamic upgrades.

In [7] Cotroneo *et al.* propose an architecture which improves the availability of web-based services, such as Web servers, FTP servers, and video-on-demand servers. Their work examines the problem from a Quality of Service (QoS) perspective and specifically targets real-time systems. Their architecture provides application developers with an alternative API which can be used to access the network instead of the typical communication libraries.

Clearly, this approach is not transparent to the developers as the provided API is used instead of the standard UNIX network socket libraries. This solution operates at the network and the operating system layers and thus cannot be applied to other platforms.

In [12] Vilas *et al.* present a work where high availability is achieved at the Web service layer. Their proposed technique introduces the notion of Virtualization. This technique creates new virtual Web services and exposes them to the clients instead of the actual ones. At the back-end, the real Web services are invoked while they are internally managed in a cluster.

The authors of this work define three requirements: detecting faulty servers, providing maintenance mechanisms for the cluster and providing mechanisms for adding and removing servers in the cluster as needed.

Virtualization is a common technique with existing and popular applications in related fields such as in web servers. The way it works in the case of Web services is by grouping one or more services inside a unique wrapper which is then published as a single, standard Web service. The clients then use this virtual Web service as if they were contacting the real one.

This approach requires that the developer defines a Virtual Web Service (VWS,) and a VWSDL document. Also a VWS engine is required to enable clustering and high availability. Depending on the complexity of the application, the VWS engine can be as simple as some specialized code in the stub or as complex as a dedicated server. Furthermore, additional techniques are needed for forming and managing the cluster.

In addition to the other works presented in this section, this work does also not provide explicit mechanisms facilitating dynamic software upgrades. In [1] Ajmani provides a thorough and comprehensive list of software upgrade techniques for distributed systems. He starts his review from Bloom's work on reconfiguration in Argus [5] and continues with recent technologies used in modern systems (such as the Red Hat OS) and also by popular services (such as in Google's infrastructure).

3.2 Evaluating existing techniques

Although all the techniques presented in this section can directly or indirectly, improve the availability of Web services, no two of them are equal with respect to their development requirements. In principle, we are interested in techniques that can be applied in general situations without any specific requirements regarding the *programming languages*, *infrastructures*, or *hardware*.

Ideally, a solution would allow automatic deployment and management of Web services and transparently improve on their availability. Apparently, such a solution should be applicable to prefabricated Web services. Furthermore, it should allow dynamic upgrades of the deployed software, preferably with minimal, if not zero, interference to the service. To the best of our knowledge, none of the presented or existing techniques fully satisfies all these requirements. In the next section we propose an architecture that can provide the basis for delivering a solution which meets all these criteria.

4 An architecture for high availability and dynamic upgradeability

This section studies the requirements for a system offering both *high availability* and *dynamic upgradeability*. Then, it proposes an architecture which is designed to meet these requirements and transparently improve on both the availability and upgradeability characteristics of prefabricated Web services. Finally, this section concludes with a discussion on the drawbacks and the benefits of the proposed design.

4.1 Requirements for high availability and dynamic upgradeability

First, the high level requirements for architectures targeting high availability are detected and enumerated. These requirements are then further complemented with additional ones targeting dynamic upgradeability, as the latter is argued to be a key requirement for improving availability.

Mathematically, availability is simply defined as the ratio of time during which the service is considered to be satisfying to the service consumer. Of course, defining when a service is satisfying is not trivial and it requires further clarification. For example, in some cases a service response of a few minutes might be acceptable, while in others sub-second responses are essential.

Detect when the service responsiveness becomes unsatisfactory. The first requirement is to detect when the service responsiveness becomes unsatisfactory to the clients. A detection mechanism must be used to detect these events and inform the appropriate components. Once deviation outside the accepted operation range is detected, a procedure is initiated which aims to resume the service. Consequently, the second requirement is to carry out the necessary actions required to restore the service normal operation within the predefined boundaries.

Manage the availability infrastructure. The second requirement is the ability to manage the availability infrastructure. For example specific architectures might need to define the order of the servers in the failover list, modify the set of servers in the cluster, or change the monitoring attributes and characteristics. In this paper we focus on the first two requirements. More management requirements are expected to be considered in future work.

The high availability technologies can be classified into those that failover on the server side, and those that failover on the client side. In the first case the classic cluster-based solution is the obvious approach, herein referred to as intra-enterprise availability. In this case a cluster of servers appears as a single server, continuously offering the service at a predefined IP address, even in the event of single server failures. This is the common case, where the clients are completely unaware of any failures or possible actions that were taken to recover the system back to fully operational mode. Naturally, in this case the clients will not be able to recover from any network outages, regardless of the cluster health.

In the second case, the client is designed to be more adaptive with regards to availability. More specifically, if a service failure is detected (and not recovered within some predefined time) the client initiates a failover procedure to another service, possibly provided by a different enterprise. We refer to this technique as inter-enterprise availability.

The mechanism for discovering and selecting a Web service in this case can be similar to that of a typical UDDI registry. More than one UDDI registries can be contacted for better fault tolerance and for a richer options pool. Additionally, this method requires specialized mechanisms embedded in the client stub and additional logic might also be necessary to ensure that the failover involves a semantically and functionally equivalent Web service. The latter is a challenging issue because

additional meta-information with regards to the service provider (e.g. pricing) might be needed when deciding on a suitable alternative service to failover to.

In the first case where the service usually runs on top of a server cluster, it is necessary to use a mechanism that continuously monitors the health of the individual servers of the cluster. In this way, any possible failures are detected before they get noticed by the clients. The failover can be performed using any of the existing methods proposed so far.

In the case of inter-enterprise availability, the detection and failover mechanisms must be embedded into the client-side. This method adds significant complexity into the clients, but has the advantage of surviving long-running network outages that prevent communication with the server side.

Support for dynamic upgrades. The last requirement we consider is the support for dynamic upgrades. By dynamic we refer to upgrades that take place at *runtime*, preferably without any service interruption. The upgrades can take place at any of the client, the server, or both sides. The following paragraphs examine the dynamic upgrade-related requirements in detail, building on results described in [2].

First a management mechanism that instructs the nodes when to upgrade must be defined. Consider for example the case where the upgrade of a service running on a cluster (i.e. for increased availability) is required. Apparently, not all the servers can be upgraded simultaneously because that would compromise the service's availability. A management mechanism can control how the servers are upgraded, so that a set of servers is consistently operational with an acceptable level of availability.

The second requirement is to provide a way to control when the servers are upgraded. Although the most straightforward solution would be to arbitrarily remove the node from the cluster and upgrade it (letting the availability infrastructure take care of the interrupted transactions) it is not an optimal one. A more appropriate solution would be one detecting an appropriate time-frame within which the upgrade would be possible without any service interruption and without breaking the consistency of the system.

The third requirement is to provide mechanisms that guarantee the normal operation of the system when nodes are running different versions of software. If, for example, the server is upgraded to support a different set of operations (e.g. specified by a different WSDL document), appropriate adaptation of the invocations is needed until the clients are also upgraded to the latest version.

The last requirement mandates a way to preserve the persistent state of servers from one version to another. If, for example, the client is in the middle of executing a long process consisting of multiple operations, it is important that the upgraded software preserves its state and continues with the next operation in the process after the upgrade is completed (rather than having to restart a large computation task). This applies to both the client and the server sides.

4.2. Smart-stubs and smart-skeletons

In order to satisfy all the requirements we have specified, we propose a skeleton architecture where components can be added and existing techniques be reused. This

architecture supports prefabricated Web services and it builds on a minimal model described in [3] and depicted by Figure 1.

In this architecture the WSDL document is used as input to specialized compilers which generate client-side and server-side proxies, typically referred to as *stubs* and *skeletons*. A different compiler is required for each of the client and the server side. The application objects can then bind to the proxies and invoke the operation defined in the WSDL documents. These proxies enable distributed communication with the use of SOAP-based messages.

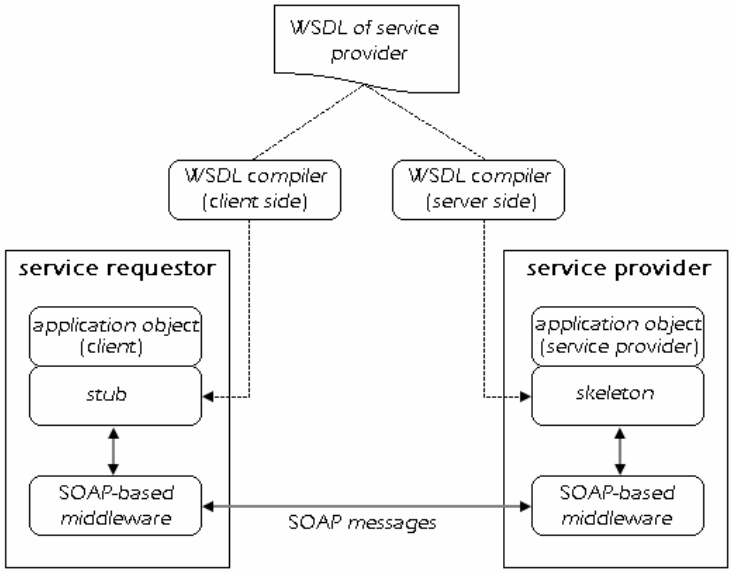

Fig. 1. Typical Web Service interaction: Based on the given WSDL document, appropriate proxy objects (i.e. the stub and the skeleton) are generated which are then used to facilitate the communication between the distributed objects by abstracting the remote objects as local.

This approach extends the idea expressed in [10], where the authors argue that the reliance on machine readable metadata is probably one of the key defining aspects of Service Oriented Architectures (SOA). In this approach, the middleware exploits the additional metadata (in the form of availability directives and preferences) to enable seamless enhancements to the overall service availability.

The presented architecture requires the dynamic generation of intelligent proxies, namely smart-stubs and smart-skeletons. These proxy components directly accept invocations from the application objects in a fashion similar to the standard Web service paradigm. The proposed architecture is depicted by Figure 2. The grayed-out areas illustrate deviations from the original architecture.

In Figure 2, the HA-related properties are used to provide information describing different aspects of the high availability-related functionality such as connection time-out, preferred failover list, etc. These properties are then encoded into the generated smart-proxies.

Special compilers (i.e. HA-aware WSDL compilers) are used for the generation of smart-stubs and smart-skeletons. In addition to providing the functionality for SOAP–based communication these proxies contain additional functionality for dealing with rerouting, blocking and adapting SOAP messages.

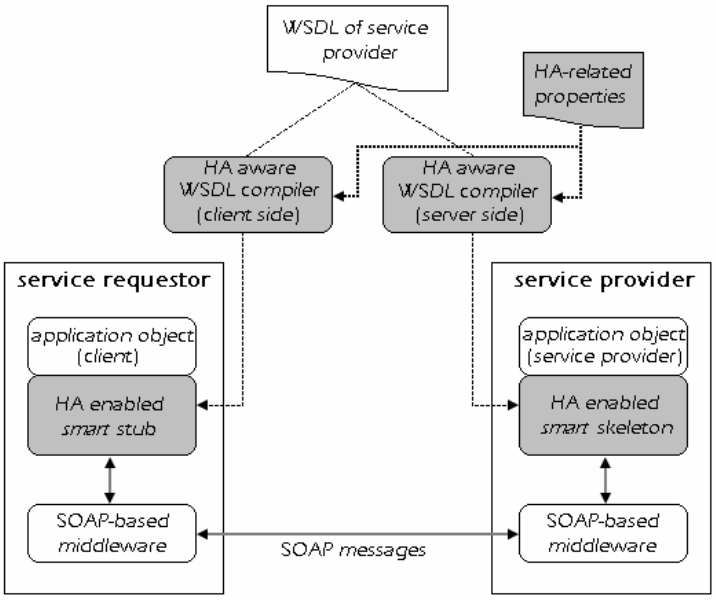

Fig. 2: Web Service interaction using smart-stubs and smart-skeletons: Based on a given set of properties describing the high availability requirements (or strategy), specialized WSDL compilers generate the smart-proxies (i.e. the smart-stub and the smart-skeleton). These objects act in a manner similar to that of the normal proxies, but additionally they incorporate specialized code which allows them to improve on the availability of the service, i.e. by enabling automatic failover and load-balancing.

The smart-stubs and the smart-skeletons are thin, automatically generated proxy components. Their role is to implement the logic required to allow automatic and seamless rerouting of SOAP messages in order to ensure high availability in the event of faults. Additionally, their role includes blocking and adapting SOAP messages in order to enable seamless dynamic upgrades of software.

To make the processing and handling of the SOAP messages transparent to the end-users, these proxies intercept the communication on both the client and the server sides and appropriately reroute, block and adapt the communicated messages.

In addition to encoding the invocations to SOAP messages and marshalling or un-marshalling the data arguments, the smart-stubs provide additional logic for handling failures on the server side (e.g. by failing over to another server). Similarly, the smart-skeletons provide functionality for blocking messages while upgrading a Web service and also for adapting messages targeting a different version of the deployed object.

4.3 Satisfaction of the requirements by the architecture.

The following paragraphs discuss how the general architecture, described here, satisfies the requirements that were detected in the previous section. First, in order to be able to discover when the service responsiveness becomes unsatisfactory, special client-side code is embedded in the smart-stub. This code allows the detection of faults and enables the failover to different service providers. Typically, this code is based on existing techniques which have a proven and trusted track in the area of fault-detection. In their simplest form these techniques usually depend on preset response-deadlines or on health monitoring systems which actively and periodically contact the servers (i.e. poke) to detect if they are responsive (i.e. healthy).

To manage the availability infrastructure the smart-proxies embed specialized functionality. Their management can be based either on static, predefined strategies encoded in the input data (i.e. in the HA-related properties), or it can based on a more dynamic and interactive scheme. The latter implies that the smart-proxies, which can serve as interception points, could be exploited to block, reroute and generally manage the operation of the Web-service from an availability point-of-view.

Additional logic might also be required to ensure the correctness of protocols such as the WS-Transaction. In particular, if the client decides to failover to another service provider while processing a business activity, specialized actions are required to ensure that suitable compensation operations are issued on the original service provider when it returns back online.

Last, dynamic upgradeability requires support by both the smart-stub and the smart-skeleton components. In particular they should both include code to address the additional, refined requirements that have been detected for enabling dynamic upgradeability.

For the management mechanism either an external, centralized coordination server should be used or special code should be embedded into the smart-stubs (or equivalently into the smart-skeletons.) Clearly, the first approach is more straight-forward from an implementation point-of-view. Embedding the code in the smart-proxies has the apparent advantage of making the system more self-reliant (but also more complicated). Finally, in simple scenarios where only a single client (or server) is upgraded, the management mechanism could be unnecessary.

For detecting when it is appropriate to perform an upgrade, specialized code should again be embedded in the implementation of the smart-stubs (or smart-skeletons). This requirement is usually related to the persistent state requirement. Suitable solutions exist which simultaneously address both. The latter usually requires that the upgraded applications provide mechanisms for enabling state persistence across different versions as well.

Finally, the concurrent support of different versions is addressed. If both the new and old versions of Web services define the same operations (i.e. they are described by the same WSDL document) then there is no need for any adaptation. If the two versions define different operations though, adaptors are required in the smart-proxies to map the old-version invocations to the corresponding operations of the new version. This is something that can be directly reused from existing solutions (i.e. designed to enable dynamic upgrades) and embedded into the smart-proxies.

Of course, this architecture is a high level overview of a skeleton system, purposely designed to intercept Web service communication at the point where invocations are applied. In this way the provided mechanisms have maximum control on the invocations and block the communication of SOAP-messages when necessary (e.g. when upgrading the Web service). Still, some issues remain to be addressed before this architecture fulfils the detected requirements.

5 Conclusions and future work

Custom solutions enabling high availability and dynamic upgradeability can be prohibitively complex and costly. In addition, they also require a combination of technologies and services such as *disaster recovery*, *consulting*, *assessment* and *management*. This paper concentrates on technical aspects of high availability and in particular on scenarios where Web service failover is used to maximize their availability. Additionally, it describes reusable techniques aiming at continuous availability during dynamic upgrades of Web services as such techniques are also required by any high availability framework.

The contributions of this paper are twofold. First, the need for availability and upgradeability is identified and existing techniques used to tackle the problem in Web service-based systems are presented. Second, a general architecture is presented which can provide the basis for systems aiming at high availability and dynamic upgradeability.

A main contribution of this architecture is that it improves on the availability of Web services, even in the event of inter-enterprise failover. As the failover mechanism entirely resides within the client this approach allows failing over to a Web service provided by a completely different entity. Also, the proposed architecture can be of benefit to prefabricated Web services as well as it only requires (re)compiling their WSDL documents to generate all that is required: the smart-stubs and the smart-skeletons. Because these smart-proxies are dynamically generated, the proposed architecture is also transparent to both the end-users, and the WSDL developers.

For the future, we plan to more elaborately define the structure and the functionality of the WSDL compilers as well as of the corresponding SOAP messages required by the protocols. Additionally, a prototype implementation is scheduled in order to evaluate the proposed architecture with the development of case study applications. Finally, related cluster management mechanisms will be examined and the use of UDDI directories for advertising and discovering alternative Web service providers will be more thoroughly studied.

6 Acknowledgements

This work was partly funded by the European Union as part of the IST MADAM project (6th Framework Programme, contract number 4169).

7 References

[1] S. Ajmani, "A Review of Software Upgrade Techniques for Distributed Systems", *http://www.pmg.lcs.mit.edu/~ajmani/papers/review.pdf*, 2002.

[2] S. Ajmani, B. Liskov and L. Shrira, "Scheduling and Simulation: How to Upgrade Distributed Systems", *9th Workshop on Hot Topics in Operating Systems (HotOS 2003)*, USENIX 2003, Lihue (Kauai), Hawaii, USA, 2003, pp. 43-48.

[3] G. Alonso, F. Casati, H. Kuno and V. Machiraju, "Web Services: Concepts, Architectures and Applications", *Springer-Verlag*, 2004.

[4] K. P. Birman, R. V. Renesse and W. Vogels, "Adding High Availability and Autonomic Behavior to Web Services", *26th International Conference on Software Engineering (ICSE 2004)*, IEEE Computer Society 2004, Edinburgh, United Kingdom, 2004, pp. 17-26.

[5] T. Bloom and M. Day, "Reconfiguration in Argus", *International Conference on Configurable Distributed Systems (CDS 1992)*, London, England, 1992, pp. 176-187.

[6] W. Cox, F. Cabrera, G. Copeland, T. Freund, J. Klein, T. Storey and S. Thatte, "Web Services Transaction (WS-Transaction)", *http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html*, 2004.

[7] D. Cotroneo, M. Gargiulo, S. Russo and G. Ventre, "Improving the Availability of web services", *22nd International Conference on Software Engineering (ICSE 2002)*, Orlando, Florida, USA, 2002, pp. 59-63.

[8] C. Evans, D. Chappell, D. Bunting, G. Tharakan, H. Shimamura, J. Durand, J. Mischkinsky, K. Nihei, K. Iwasa, M. Chapman, M. Shimamura, N. Kassem, N. Yamamoto, S. Kunisetty, T. Hashimoto, T. Rutt and Y. Nomura, "Web Services Reliability (WS-Reliability) version 1.0", *http://www.oracle.com/technology/tech/webservices/htdocs/spec/WS-ReliabilityV1.0.pdf*, 2003.

[9] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management", *IEEE Transactions Software Eng*ineering, 16 (1990), pp. 1293-1306.

[10] N. K. Mukhi, R. Konuru and F. Curbera, "Cooperative Middleware Specialization for Service Oriented Architectures", *13th International World Wide Web Conference (WWW2004)*, New York, NY, USA, 2004, pp. 206-215.

[11] P. Oreizy, N. Medvidovic and R. N. Taylor, "Architecture-Based Runtime Software Evolution", *20th International Conference on Software Engineering (ICSE 1998)*, IEEE Computer Society, Kyoto, Japan, 1998, pp. 177-186.

[12] J. F. Vilas, J. P. Arias and A. F. Vilas, "High Availability with Clusters of Web Services", *6th Asia-Pacific Web Conference (APWeb 2004)*, Springer-Verlag, Hangzhou, China, 2004, pp. 644-653.