A Centralized Self-Healing Architecture as Support to Web Services Applications

Francisco Moo-Mena, Juan Garcilazo-Ortiz, Luis Basto-Díaz, Fernando Curi-Quintal, and Fernando Alonzo-Canul

Universidad Autónoma de Yucatán – Facultad de Matemáticas
Periférico Norte km 33.3, Merida, Mexico
{mmena, gortiz, luis.basto, cquintal}@uady.mx, fernandoalonzo.0@gmail.com
Paper received on 14/08/10, Accepted on 26/09/10

Abstract. Web Services (WS) technology proposes tools to implement complex distributed systems. It allows creating networks of heterogeneous and cooperative WS on applications. Due to the complexities related to the dynamics of this kind of system (new WS could come in and come out of the system), applications are exposed to several failure points in both components and connections. Current efforts in WS literature offer a supporting infrastructure for this problem; most of them are oriented to repairing, mainly on application's workflow level. In this paper, we expose our ideas about an infrastructure based on Quality of Service (QoS) analysis of WS-based applications. Our approach considers a complementary strategy, supported by the adaptability of the application's architecture. This strategy is planned to prevent and respond to QoS degradation in WS-based applications, trying to enhance their performance. Our solution is oriented to discover new approaches for the three main stages identified in the related literature for self-healing infrastructure: Monitoring, Diagnosis and Recovering. In order to show our strategy effectiveness, we implemented it in the access to a Digital Library based on WS.

Keywords: Self-Healing System, QoS, Box-Plot Diagram, Web Services Application, Digital Library.

1 Introduction

Currently, evolution of WS technology offers a great potential in order to develop complex applications in different contexts, such as enterprises, industry, government, or home. Implementing these applications requires creating a network of cooperative WS. Despite the fact that WS technology deals with component's heterogeneity, these WS networks present the inconvenience to be open (exposed to everyone) and highly dynamics (new WS can come in and others come out). To deal with both conditions is an actual challenge.

In a general context, this set of problems can be extrapolated to the complex distributed applications development. In this sense, IBM presented, early this decade, an initiative named Autonomic Computing [1], which approaches these problems proposing four axis of study:

1. *Self-configuring*, deals with components and systems reconfiguration by defining high-level politics. *Self-healing*, deals with auto-detection, diagnosis, and recovering in hardware and software problems.
2. *Self-optimizing*, deals with parameter auto-adjusting at service level.
3. *Self-protecting*, deals with system protection against attacks and intrusions.

Our work focuses mainly on self-healing systems whose critical aspects are: a) mechanisms for system's health maintenance process, b) system's failure detection, and c) system's recovery processes [2].

In general, self-healing process consist of three stages, [3]:

1. *Monitoring*. This stage is responsible for monitoring and recording information about the status of system, activity in order to maintain the reliability level and quality of service.
2. *Diagnosis*. Information collected is examined, and the cause of malfunction or failure is determined. In this stage, a system assessment is done using a reference model.
3. *Recovery*. This stage is responsible for maintaining or restoring the correct state of the system.

Attending to these problems, based on IBM's initiative, several projects have emerged aiming to create a supportive infrastructure for applications. Some examples of these efforts are [4], [5], [6].

This document describes an approach proposing a strategy for WS-based applications with auto-adaptation capabilities. This strategy is based on implementation of architectural reconfiguration techniques, such as WS duplication and substitution defined on previous work [7], [8]. Decision about when to apply an architectural configuration action is determined by defining, monitoring and diagnosing QoS parameters by using statistical techniques based on box-plot diagrams. The main goal in diagnosis techniques is to prevent QoS degradation when executing WS-based applications.

Typically, recently developed WS applications are oriented to business and services areas, for example, travel agency systems and "production chain" process oriented systems, such as client/supplier/producer systems. Thus, the implementation of our strategy on a digital library application represents another innovative proposal.

The term "digital library" has been used to define a large digital information storage accessed through computer networks [9]. As in traditional libraries, a digital library serves as a knowledge archive of different subjects. Since information can be filed in different formats, a digital library would contain text, audio, image, animation and video files. Digital libraries posses features like creating digital documents, indexing and searching, management and access control, personalization, information distribution, suitable interface to deliver results, etc.

The rest of this document is organized as follows: the second section presents some important works with regard to self-healing systems. The third section defines our general strategy for designing and implementing a self-healing infrastructure for WS-based applications. The fourth section shows some important results of this work. Finally the last section describes the conclusion and future perspectives for this work.

2 Related Work

In the context of Autonomic Computing some important works following a self-healing approach are:

The Bio-Networking Architecture [10] is a paradigm as well as a middleware that enables the construction and deployment of scalable, adaptive, and survivable/available applications. In this work, applications are constructed using a collection of autonomous mobile agents called cyber-entities, and the Bio-Net Architecture platforms (execution environments and support services for the cyber-entities). It is based on principles and mechanisms used by biological systems to adapt to changing environmental conditions, like colonies of ants and bees.

The Hydra project [11] is an Integrated Project that develops middleware for Networked Embedded Systems. The Hydra project is co-funded by the European Commission. It is a middleware that allows developers to incorporate heterogeneous physical devices into their applications by offering easy-to-use-Web service interfaces for controlling any type of physical device irrespective of its network technology such as Bluetooth, RF, ZigBee, RFID, WiFi. Hydra incorporates means for Device and Service Discovery, Semantic Model Driven Architecture, P2P communication, and Diagnosis.

The CODA project (Complex Organic Distributed Architecture) [12], includes a means monitoring and controlling objectives to allow the enterprise to evolve with certain degree of autonomy. The architecture includes concepts and principles of Self-organization, Self-regulation toward an intelligent architecture. CODA is organized into five functional layers, each with a storage component and an intelligent component:

1. *Operation:* This layer manages data and connects simple linear data from databases in different locations.
2. *Monitoring Operation:* This layer ensures internal monitoring.
3. *Monitors Monitor:* in this layer operations are monitored in terms of an external monitoring.
4. *Control:* layer that has the ability to learn about the behaviour, trends and predictions.
5. *Command:* the highest-level layer that is responsible for recognizing threats and opportunities.

In the area of Web services and within this category is the WS-DIAMOND (Web-Service Diagnosability, Monitoring & Diagnosis) project [13], which considered the study and implementation of methodologies and solutions by creating self-healing Web services able to detect abnormal operation such as the inability to respond to service's requests or failure to achieve a level of QoS stipulated as a requirement. As well as to restore service from failures by restructuring or reconfiguring network services. WS-Diamond also provides methodologies for the design, support and service running mechanisms to ensure monitoring, diagnosis and fault recovery at runtime.

3 Self-healing Architecture

Our architecture is oriented to Web service's interaction (interaction between two WS). There are always two main actors, the consumer (service requester) and the provider, the main scenario where the architecture works is divided into two phases; the first one is used to collect the information regarding to Web service's interaction and is represented by an interceptor, and the second one is in charge of processing the data collected and applying the corresponding self-healing mechanism.

3.1 Interceptors

Interaction between two WS is very simple; there is a consumer and a provider (see figure 1). We added a third actor, the interceptor located in both the WS consumer and the WS provider in order to register the following records:

- T1: *Service Request's start time*. Time at WS consumer sends the request.
- T2: *Service Request's end time*. Time at request arrives to WS provider.
- T3: *Service Response's start time*. Time at WS provider sends the response.
- T4: *Service Response's end time*. Time at WS consumer receives the response.

All the four monitoring records (T1, T2, T3, and T4), help us to define the QoS parameters such as Computation Time, Requesting Time, Responding Time and Communication Time, those parameters are calculated as follows:

- QoS1:= T3 - T2 : *Computation Time.*
- QoS2:= T2 - T1 : *Requesting Time.*
- QoS3:= T4 – T3 : *Responding Time.*
- QoS4:= (T4 - T1) - QoS1 : *Communication Time.*

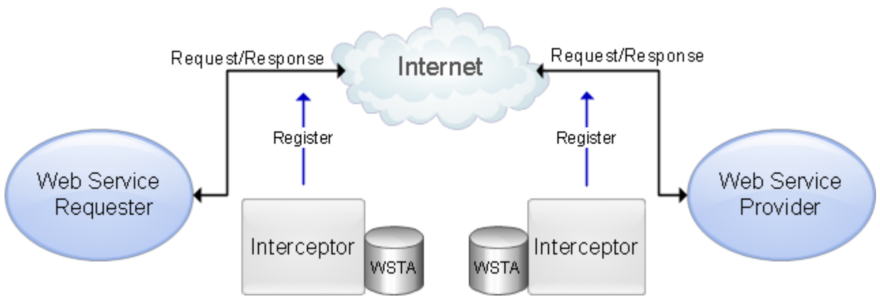

Fig. 1. Interactions between Web services.

In order to use our self-healing architecture the WS consumer needs to implement our Client API, which provides all the interfaces required to send a service request to provider and receives its response. With Client API, client uses the self-healing benefits in a transparent way. Consumers use the Client API in the following way:

1. Create a Client instance.
2. Create a Request instance.
3. Define Web service and operations desired on request instance.
4. Associate the request instance to client instance.
5. Invoke client instance and execute function.
6. T1 is registered and sends transaction to provider.

7. Provider interceptor registers T2 measure.
8. Provider processes Request.
9. Interceptor registers T3 measure.
10. Response arrives to Consumer.
11. T4 measure is registered.

3.2 Self-healing Components

The self-healing core architecture (see figure 2) shows the main components: monitoring, diagnosis, recovering and interceptors.

Also, there are three elements used as information repositories, two of them related to QoS. The Parameter Repository (PaRe) stores all the QoS measures registered by interceptors. The Service Level Parameters (SLP) represents the optimal measures to be accomplished in every Web service's interactions, that is, the contract agreement. The last information repository component is the WS Table Access (WSTA). Conceptually, it has the service information: identification, description and the WS in charge of response to the service request.

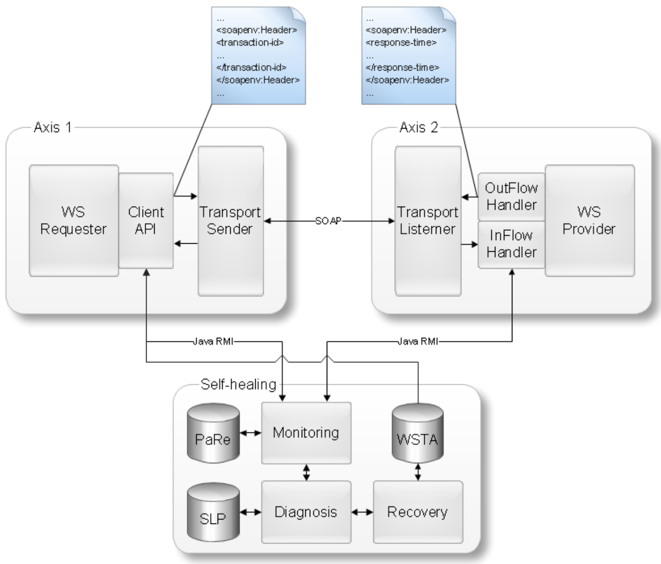
```
...
<soapenv:Header>
<transaction-id>
...
</transaction-id>
</soapenv:Header>
...
```
```
...
<soapenv:Header>
<response-time>
...
</response-time>
</soapenv:Header>
...
```

Fig. 2. Self-healing core architecture.

3.3 Monitoring Module

Monitoring module is the only interface between interceptors and architecture's core, being its main objective to collect data between interceptors and store them into PaRe. Interaction begins when a consumer sends a request service to provider and ends when the consumer receives a request response. At the same time if a failure is detected, immediately monitoring module will be alerted, (see figure 3).

Monitoring module implementation uses RMI as a way to receive data from interceptors and sends data to persistence databases through Hibernate [14].

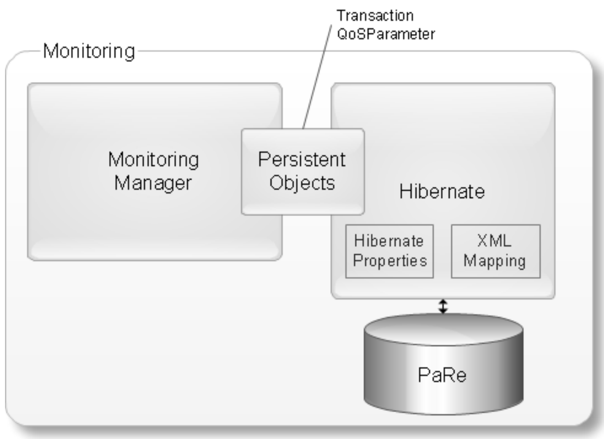

Fig. 3. Monitoring module.

3.4 Diagnosis Module

As we mentioned, monitoring module collects QoS data and stores them into PaRe (Parameter Repository). All data collected are sent to Diagnosis Module (referenced by DIMOD at the rest of paper) if any of the following scenarios occurred:

1. A Web service does not response.
2. Monitoring module collects one hundred transactions.

Based on SLP information, data collected by monitoring module are analyzed by DIMOD, in order to find out architecture behaviour indicating a degraded state; it is presented when DIMOD detects an amount of outliers. We used a statistic model as a main tool to know when application behaviour presents any of the defined states (Optimal, Good or Degraded). In order to define the model, a BoxPlot [15], [16] method was suggested; it is based on a graphic box, which represents a data distribution based mainly on three measures: inferior, median and superior quartiles. On a BoxPlot diagram outliers are very easy to identify being outside of the box. Table 1 shows the variables considered and used by BoxPlot, all of them are based on information collected by monitoring module.

Table 1. Variables considered by BoxPlot.

Variables	Description
N	Total of transactions processed.
x_i	External border for QoS_i, $x_i = Q_3 + 3(IQR)$.
m_i	Amount of outliers values, where $QoS_i >$
p_i	% of outliers values, $p_i = m_i * 100 / n$.

Previous table shows N as the total of transactions that diagnosis model analyzes. Every transaction has four QoS parameters (QoS1, QoS2, QoS3 y QoS4), used to calculate external borders.

Table 2. QoS level agreement.

Criteria	**pi = 0%**	**pi < 5%**	**pi > 5%**
QoS level	**Optimal**	**Good**	**Degraded**

According to data expressed on table 2, a very optimistic scenario is defined, when no outlier is registered, we do not expect this scenario all time. With the percentage of outliers values less than 5% and greater than 0% a good scenario is defined, our main goal is keep a QoS level between good and perfect values. Finally a degraded state is defined when that percentage is greater than 5%, in this case the recovering module will apply a reconfiguration strategy in order to guarantee the QoS agreement.

As mentioned previously, the monitoring module enables the diagnosis whenever one hundred transactions are completed or when an error occurs, and is necessary an immediate reconfiguration action.

The diagnosis process verifies the compliance with the criteria described above, calculating the percentage of outliers, from SLP values, for QoS parameters of the transaction whose identifier is in the range defined by startIndex and endIndex. While all the QoS parameters meet the limits defined by the SLP, the diagnosis module will not raise any recovery action and the range of transactions will increase, setting a new endIndex.

If a QoS parameter does not meet the required limit, then the diagnosis module alerts the recovery module to perform the appropriate reconfiguration action and defines a new range of diagnosis, ruling out the QoS parameters of transactions previously analyzed.

If the monitoring module records that a transaction was not completed successfully because a service provider is out of line, it notifies immediately to the diagnosis module, and to the recovery module, using the method activeSusbtitution. A substitution action is applied immediately to redirect the request to another provider.

3.5 Recovery Module

When diagnosis module detects a QoS degradation, recovery module is alerted in order to stabilize the application. When a WS is down, a new one replaces it immediately. This is considered a panic error scenario on our self–healing architecture.

We defined two recovery actions and their implementation will depend on the degree of degradation of the QoS, based on the percentage of outliers reported by the diagnosis module (see table 3).

Table 3. Recovery actions.

Degradation degree	**$5\% < p_i \leq 10\%$**	**$p_i > 10\%$**
Recovery action	Duplication	Substitution

Duplication actions divide all WS requests between the current WS and a new one that offers the same operation, in order to meet the service quality agreement. This action will keep active these two services in the WSTA instance.

The action of substitution replaces the current service for a new one. It also includes deletion of the previous service from the WSTA instance.

All service information is stored on WSTA table, containing more than one service, with similar operations. The recovery module actives a service, as a result of any reconfiguration actions described above. This module also uses Hibernate as a middleware to manage persistence information.

4 Results

In order to test our approach based on BoxPlot diagrams, we implemented a system with all the necessary components to monitor application performance, diagnose application's health and apply reconfiguration approaches into architecture. We worked with the PDLib digital library application [17]. We adapted this application in order to enable WS interfaces.

There are many services available on a digital library, we chose listDocumentation service, used to show all documents stored in a database, as a main service in our test. We launched 10,000 requests to the application, trying to obtain enough data to validate our diagnosis module implementation.

After execution of testing transactions, we got the results shown in the following three tables referenced as table 4. Values are expressed in milliseconds.

Table 4. Experimental results.

	Frequency	Mean	Median	Mode
QoS1	10000	27.9598	22.0	20.0
QoS2	10000	26.0621	20.0	17.0
QoS3	10000	7.3885	6.0	5.0
QoS4	10000	33.4531	26.0	26.0

	Standard deviation	Minimum	Maximum	Range
QoS1	28.9272	14.0	1061.0	1047.0
QoS2	32.2210	16.0	1154.0	1138.0
QoS3	11.9261	4.0	377.0	373.0
QoS4	33.5988	21.0	1160.0	1139.0

	First Quartile	Second Quartile	Third Quartile	Interquartil e Range
QoS1	20.0	22.0	23.0	3.0
QoS2	17.0	20.0	21.0	4.0
QoS3	5.0	6.0	6.0	1.0
QoS4	23.0	26.0	27.0	4.0

Experimental results in table 4 show QoS limit values for each quartile. Based on third quartile measures (Q3) we calculated external borders, xi = Q3 + 3(IQR), for each QoS parameter. Table 5 shows extreme borders values.

Table 5. QoS external borders values.

	xi = Q3 + 3(IQR)	External Borders
QoS1	x1 = 23.0 + 3(3.0)	x1 = 32.0
QoS2	x2 = 21.0 + 3(4.0)	x2 = 33.0
QoS3	x3 = 6.0 + 3(1.0)	x3 = 9.0
QoS4	x4 = 27.0 + 3(4.0)	x4 = 39.0

Each value bigger than its external border is considered as an outlier. Based on table 5 outliers values for QoS1 will be those bigger than 32 ms, bigger than 33 for QoS2, and so on. All values greater than its corresponding external border are considered as unexpected values.

Following figure 4 is a BoxPlot diagram showing QoS results and we can see that most of the measurements are skewed at the right and near of the third quartile. Few measurements are high for the four QoS parameters having 9.46%, 9.15%, 7.67%, and 10.69% respectively of unexpected values. These values are stored in PaRe in order to define the acceptable limits.

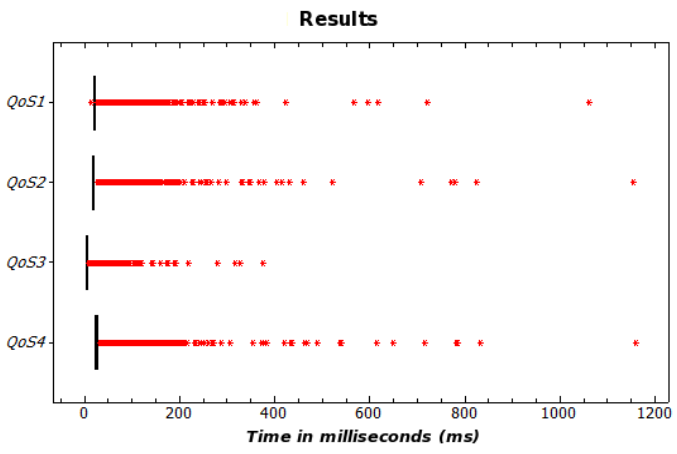

Fig. 4. BoxPlot diagram for QoS registers.

If an application does not accomplish with the QoS level agreement, diagnosis module will detect QoS degradation and recovery module will be alerted in order to stabilize the application. When a WS is down, immediately, is replaced by a new one. This is considered a panic error scenario on our self–healing architecture.

Another test measured the self-healing impact on application performance. So we repeat the previous test with (TestCase1) and without (TestCase2) self-healing architecture having that the values for each QoS parameters vary only in a few milliseconds, which means that the cost-benefit of using our architecture is favourable as they have all the benefits of self-healing and the cost is relatively low. The biggest difference we can notice is in the parameter QoS1, with the increase of 1.0901 ms on Mean, because in this time period are carried out most of the additional processes required to ensure the QoS.

5 Conclusion

Each module in our self-healing architecture carries out specific tasks and has well-defined interfaces through it communicates with the other modules. The novel points of our diagnosis model are the inclusion of the QoS parameters and the construction of a statistic model based on BoxPlot diagrams. Based on that, we apply reconfiguration actions. Results showed how diagnosis module obtains right conclusions about system health, sending correct information to recovery module.

In order to measure performance and stability of our architecture, two test scenarios were designed by supporting functionality of a digital library application. The first scenario consisted of applying the architecture to the digital library system and calculating statistical measures from the results. The second scenario is similar to the first one, but the main difference is that the complete architecture was deactivated in order to compare the obtained results previously and to calculate the additional processing time that is generated when using our architecture.

The accomplishment of these tests threw favourable results, since when doing the comparison between the obtained results from both scenarios we could observe that when deploying our architecture it affected the application results in an insignificant way, and the benefits that can be obtained are very important. This means that there are compensation between self-healing features implemented against cost and performance.

As a future work, we consider extend our statistic model exploring other diagnosis approaches like the definition and implementation of semantic techniques based on ontologies. Furthermore, diagnosis module will determine precisely failures, and recovering module will be able to apply a better reconfiguration action.

With respect to the digital library application, the development of new interfaces is considered in order to provide more services related to this application.

Finally, developing test scenarios where architecture modules are lesser centralized is currently considered.

Acknowledgments. This project is developed under financial support from PROMEP and UADY.

References

1. Kephart, J., and Chess, D. (2003). The vision of autonomic computing. Computer, 1(36):41–50.
2. Ghosh, D., Sharman, R., Rao, H. R., & Upadhyaya, S. (2007). Self-healing systems - survey and synthesis. ACM Digital Library, 42, 2164-2185.

3. Ben Halima, R., Drira, K., and Jmaiel, M. (2008). A QoS-Oriented Reconfigurable Middleware for Self-Healing Web Services. IEEE International Conference on Web Services, ICWS '08. Pages 104-111, Beijing, China.
4. Garlan, D. and Schmerl, B. (2002). Model-based adaptation for self-healing systems. In Proceedings of the first workshop on self-healing systems, WOSS ’02, pages 27–32, New York, NY, USA. ACM Press.
5. Wile, D. and Egyed, A., (2004). An externalized infrastructure for self-healing systems. In Proceedings of the Fourth Working IEEE/IFIP Conference on Software
6. Gurguis, S. A., Zeid, A., (2005). Towards autonomic web services: achieving self-healing using web services. In DEAS ’05: Proceedings of the 2005 workshop on design and evolution of autonomic application software, pages 1–5, NY, USA, ACM Press.
7. Moo-Mena, F. J., Drira, K., (2007). Reconfiguration of web services architectures: a model-based approach. In Proceedings of the Twelfth IEEE Symposium on Computers and Communications (ISCC’07), pp. 357–362, Aveiro, Portugal. IEEE Computer Society.
8. Moo-Mena, F. J., Drira, K., (2007). Modeling architectural level repair in web services. In Proceedings of the 3rd International Conference on Web Information Systems and Technologies (WEBIST’2007), pages 240–245, Barcelona, Spain.
9. Association of Research Libraries (1995). Definition and purposes of a digital library. http://archive.ifla.org/documents/libraries/net/arl-dlib.txt
10. Wang, M., Suda, T. (2000). The bio-networking architecture: A biologically inspired approach to the design of scalable, adaptive, and survivable/available network applications. Tech. Rep. 00-03, University of California, Irvine, California, USA.
11. Eisenhauer, M., Rosengren, P., and Antolin, P. (2009). A Development Platform for Integrating Wireless Devices and Sensors into Ambient Intelligence Systems. 6th Annual IEEE Conference on Communications Society. Pages 1-3, Rome, Italy.
12. Ribeiro, G. and Karran, T., (2001). An Object-Oriented Organic Architecture for Next Generation Intelligent Reconfigurable Mobile Networks, doa, pp.0031, Third International Symposium on Distributed Objects and Applications (DOA’01), 2001.
13. Console, L., et al. (2008). At your service: Service Engineering in the Information Society Technologies Program - WS-DIAMOND: Web Services - DIAgnosability, MONitoring, and Diagnosis. Editor E. di Nitto, A-M. Sassen, P. Traverso, and A. Zwegers, MIT Press.
14. King, G,, Bauer, C., Rydahl-Andersen, M., Bernard, E., and Ebersole S. (2010). Hibernate Reference Documentation. Copyright © 2004 Red Hat, Inc.
15. Scheaffer, R.L. and McClave, J.T (1994). Probability and Statistics for Engineers. Duxbury Prees, 4th Edition.
16. Mendenhall, W., Beaver, R. J., and Beaver, B.M (2008). Introduction to Probability and Statistics. Barnes & Noble, 559 p.
17. Alvarez, F., García, R., Garza, D., Lavariega, J., Gómez, L., and Sordia, M., (2005). Universal access architecture for digital libraries. In Proceedings of the Conference of the Centre For Advanced Studies on Collaborative Research, pages 17–20, Richmond-Hill, Ontario, Canada.