Journal of Web Engineering, Vol. 11, No. 1 (2012) 079-092
© Rinton Press

PREDICTIVE SELF-HEALING OF WEB SERVICES USING HEALTH SCORE

MOHSEN SHARIFI[*] SOMAYEH BAKHTIARI RAMEZANI

School of Computer Engineering, Iran University of Science and Technology
{msharifi,bakhtiaris}@iust.ac.ir

AMIN AMIRLATIFI

Auton Solutions
amin.amirlatifi@autonsolutions.com

Received April 25, 2010
Revised November 24, 2011

Existing self-healing mechanisms for Web services constantly monitor services and their computational environment, analyze system state, determine failure occurrences, and execute built-in recovery plans (MAPE loop). We propose a more pro-active self healing mechanism that uses a multi-layer perceptron ANN and a health score mechanism to learn about the occurrences of failures or quality of service degradation in advance, without requiring modifications to the framework of services used by applications. Highest score is assigned to the system upon start and is degraded during system execution whenever a service fails to operate or the time-to-leave (TTL) of the client side requests increases. Application of the proposed mechanism to a set of vehicle tracking Web services decreased the probability of out of service occurrences by 70% and increased system quality of service by 13%. The overhead of the mechanism was nearly 3% and negligible, whilst TTL for a request from the client side decreased by 20%.

Keywords: Web Services, Predictive Self-Healing, Failure Prevention, Failover, Quality of Service, High Availability, Health Score
Communicated by: M. Gaedke & P. Fraternali

1 Introduction

Increased tendency towards Web services and the wide range of services offered through them necessitates the need for reliable services, both with regard to high availability and quality of service. For this reason, many researchers have tried to keep Web services in healthy state for prolonged periods using some self-healing mechanisms [1, 2, 3, 4, 5]. They however do this after the occurrences of faults, which is too late and too costly to detect and recover faulty services. Furthermore, faulty services remain unavailable until they are completely fixed.

In this paper, we propose a predictive self-healing mechanism for early detection and treatment of faulty Web services. The goal is to maintain quality of services and to provide high availability of services in the presence of faults in Web services.

Another weakness of existing proposed healing mechanisms [1, 2, 3, 4, 5] is that they require modifications to the framework of Web services that are used by applications; this is to say that each of these mechanisms is highly dependent on the characteristics of a single framework for Web services.

We have used rather general characteristics of Web services in our proposition for early detection and fixing of faulty services, quite independent of any special features of any framework for Web services. This makes our self-healing mechanism to work with any existing or future framework for Web services. This was achieved through monitoring of the trend of system and Web service parameters and detecting anomalies based on these trends.

The rest of paper is organized as follows. Section 2 presents some notable related works. Section 3 presents our architecture and Section 4 presents our implementation. Section 5 argues the validity of the proposed mechanism, and Section 6 concludes the paper.

2 Current Practice and Research

Paul Horn first brought up the idea of autonomous computing in 2001 to address the ever-increasing complexity of software systems [6]. A true and comprehensive autonomous system, per definition, is comprised of four characteristics: self-configuring, self-healing, self-optimizing, and self-protecting [7, 8]. Since then, researchers have taken many important aspects of Web services and self-healing issues into consideration in their researches, [9, 10, 11, 12, 13]. Automated service discovery and composition [9, 11, 12, 13], and automated validation of the resulting composition has been long sought [14, 15]. Recent works have focused on the quality of service aspects [16], and targeted to address the full range of autonomic computing in general [16, 17] and self-healing in particular [18, 19].

The work by Chan *et al.* [3] is one of the works most related to ours. They have proposed a cycle for self-healing of Web service composition based on MAPE loop that includes monitoring, analysing, planning, and plan execution for recovery. Re-planning is required when severity of fault is lowered. It is not however clear at all who should run these operations at each cycle. One of the disadvantages of their approach is the lack of prediction of fault or error in the system. Downside of this issue is that faults may be detected when it is too late to recover quality of service. In addition, the proposed cycle is limited to Web service compositions and cannot be used for standalone Web services.

Although very few attempts have been made to add self-healing to existing programs, most of them come into play only after the fault has occurred unlike our work wherein fault prevention is sought. Fuad and Oudshoorn [20] have tried to transform existing programs into self-healing entities by injecting their code into an existing application and encapsulating the Java run-time code inside an extra layer of exception catching. They have tried to fix transient faults by looking at a preset table of rules.

Many authors [1, 2, 3, 4, 5, 17, 18, 21, 22, and 23] have proposed monitoring and self-healing mechanisms for self-diagnosis and self-healing that address different types of Web services. They all share the same conceptual difference with our work, as we predict the occurrence of the fault and try to

prevent it before happening, while others try to correct the fault after the system has already suffered. Noui-Mehidi [24] has suggested a predictive algorithm for monitoring the performance and the overall health of the system and taking corrective actions before Web service experiences faults. His approach is very similar to our work in the context of predicting the fault, but unlike his work, our work has a wide application range that can include existing and new Web services and does not require user intervention.

Alonso *et al.* [25] have tried to build an accurate model of the system that is suffering from transient failures. They have compared linear regression and decision tree algorithms and concluded that M5P decision tree is the best option for modeling the behavior of the system under random injection of memory leaks.

Naccache *et al.* [5] have proposed a framework for developing an autonomic self-healing portal system that relies on the notion of differentiated services. This framework supports existing Web services through a lightweight wrapper. It can also be used as a basis for developing new Web services, but due to the emphasis on Ajax-based systems and the lack of failure prediction, the applicability of this framework is limited.

Yan *et al.* [26] have made another attempt in this regard. They have tried to lay grounds for a monitoring and diagnosis tool for Web services. They have tried to make it possible to detect abnormal situations, identify the causes of these abnormalities, and to decide on recovery actions. Due to their abstract way of modelling components, their approach cannot satisfy scalability issues.

There are several ways of implementing self- healing mechanisms. For example, Mostefaoui *et al.* [4] have used Aspect Oriented Programming (AOP) for design and development of a self-healing Web service. As another example, Ben Halima *et al.* [2] have proposed a self-healing framework for quality of service management in service-based Web applications. They have implemented their framework on top of reflective programming libraries. Their framework mainly relies on intercepting and handling contents of SOAP-level communication messages.

Due to the nature of Web services, faults that occur in Web services are usually repetitive. Pernici *et al.* [1] have proposed a methodology and a tool for automatic learning of repair strategies for Web services so that the best repair strategy can be selected. They show that their autonomous approach can compare features of a new fault with those of previously detected faults and classify the fault based on its persistence level. Their approach cannot determine fault origins, which can help to remove or resolve the conditions that determine faults. We have tried to address this shortcoming and repair the service structure to prevent the occurrence of faults. In addition, we have introduced and improved a weighting mechanism for use at run time, as opposed to the weights that Pernici *et al.* [1] use at design time for identification of faults, by coupling it with an artificial neural network, after normalizing values of different parameters obtained from Web service and the working environment.

However, it seems none of the researchers has tried to use Health Score or a similar metric that uses TTL and Responsiveness of Web service to predict the occurrence of faults in advance. Instead, they have tried to put forward mechanisms for healing the faults after they occur, which includes downtime of service, whereas preventive nature of our work eliminates any downtime. We try to predict and prevent the occurrences of faults beforehand or at least initiate a self-healing process in cases where the faults cannot be prevented.

3 Architecture of the Proposed Mechanism

The proposed mechanism is composed of two phases, namely, the knowledge acquisition phase, and the execution phase. In this work, we have added a new learning phase to the usual self-healing frameworks, MAPE loop, as compared to execution phase of usual frameworks. Figure 1 shows the architecture of our proposed mechanism.

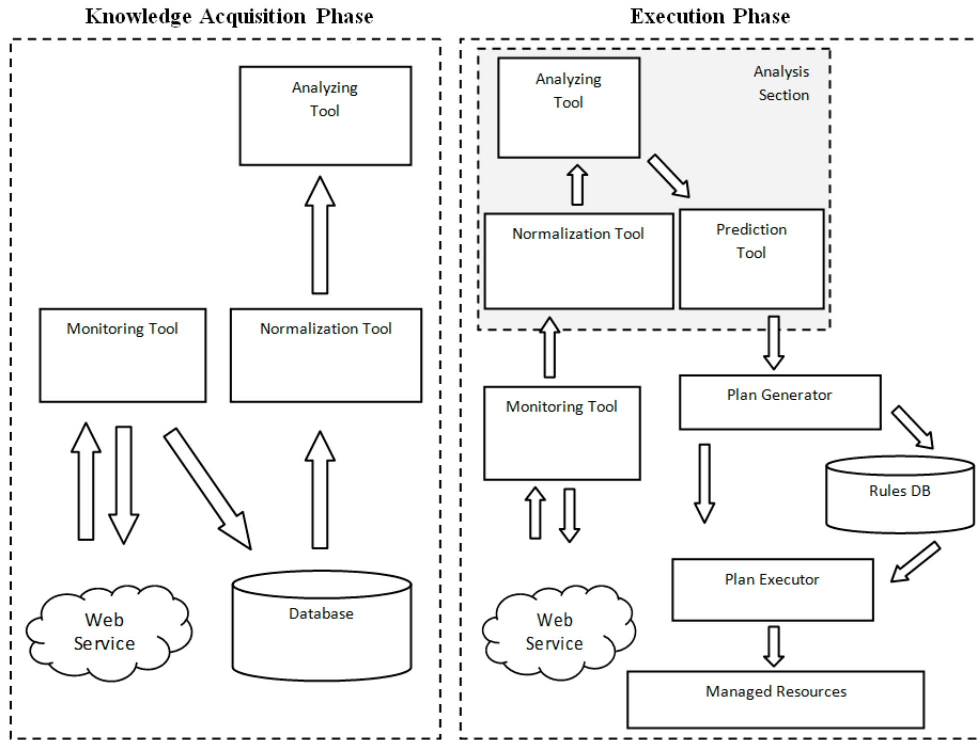

Figure 1: Architecture of our predictive self-healing mechanism

3.1. Knowledge Acquisition Phase

The knowledge acquisition phase is considered as a new addition to current self-healing solutions and is used to determine the cause of faults in the system, based on important parameters of Web services themselves and the working environment that hosts Web services. This phase tries to increase the precision of fault prediction by comparing healthy and unhealthy states of the system. Since events are logged with their date and time stamps, we will have the chain of events in a timeline, which enables us to locate scenarios that lead to fault.

In this phase, the effects of Web services on the working environment as well as the effects of the environment on Web services are gathered and analysed for a given period, e.g., one month. Based on the context of Web services, the length of this period can be altered.

A Monitoring Tool that imposes little load on the system and does not consume much CPU or system resources will store values of different parameters in adjustable data logging intervals, in a

database. Table 1 shows the working environment and Web service parameters that are monitored by the Monitoring Tool.

Table 1: Web service and working environment parameters

Web Service Parameters		**Working Environment Parameters**
Service Dependant	**Machine Dependant**	
No. of active clients	CPU usage	CPU usage
No. of active TCP/UDP listeners	RAM usage	total installed RAM
TTL	I/O	available RAM
	network activity	No. of active processes
	response time	high demand process (CPU, RAM, I/O, Network usage)
		I/O
		network activity

Time To Leave (TTL) is one the parameters that is critical to the evaluation of quality of service of Web services; TTL for a Web service request is defined as the time it takes for the Web service to process a request from its arrival at the Web service to the time it is marked as handled (successfully or unsuccessfully). We have assumed that we have the means to measure TTL.

Selection of adequate data logging interval plays a crucial role in effective data logging. Assuming that a Web service or the working environment has approached a critical state or confronted a fault, a long logging interval may result in this event to be unnoticed by the monitoring tool. On the other hand, short data logging intervals create considerable workload on the system. As a result, the best practice is to assign a variable data logging interval that varies by the workload imposed on the Web service or the working environment.

Since recorded parameters are responsible for changes in the states of a Web service from healthy to unhealthy state or vice versa, all of their effects are taken into account and a weight that denotes their share in state change is assigned to each parameter.

In this mechanism, we have mapped observed values for each parameter, to a value in the range between zero and one. For each parameter, the normalizing coefficient is defined as the parameter value multiplier that maps the parameter $0 \leq \alpha * \beta \leq 1$ value to this range, where β is the value of each parameter and α is defined as the normalizing coefficient.

Instead of looking at Web service health on a two-state scale of healthy or unhealthy services, we have defined a measure of Health Score (HS) in this mechanism that depends on TTL and responsiveness of the Web service. HS has an inverse relationship with normalized TTL, meaning that normalized TTL of 0 results in Health Score of 1 and normalized TTL of 1, and that a non-responsive Web service results in Health Score of 0. Table 2 shows the relationship between TTL, Web Service Responsiveness State, and Health Score.

Table 2: Health scores based on TTL and responsiveness state

TTL Range	Responsiveness State	HS
1	Non-Responsive	0
1	Responsive	0.25
0.75 – 1	Responsive	0.25
0.5 - 0.75	Responsive	0.5
0.25 - 0.5	Responsive	0.75
0 - 0.25	Responsive	1

In order to be able to describe the state of Web service better, we have defined a new definition: Overall Index (OI). OI is an indicator of the current state of the Web service. Generally, Web services are described as either healthy or unhealthy. In order to provide a better measure, we assume that the overall index is not simply limited to healthy or unhealthy states, but it also includes other states such as somewhat healthy, intermittent and somewhat unhealthy states (Table 3).

Table 3: Overall index and quality of service based on health score

QoS	OI	HS
Out of Service	Unhealthy	0
Low	Nearly Unhealthy	0.25
Acceptable	Intermittent	0.5
Good	Nearly Healthy	0.75
High	Healthy	1

Analysis of normalized parameter values derived from the working environment, e.g., CPU usage, disk read/write, and network traffic, alongside normalized Web service parameters, such as TTL, RAM usage and CPU usage at different times and in sequential order, together with the analysis of the trend of Web service health change from healthy to unhealthy states, provide valuable information about the causes of faults allowing to accordingly assign a reasonable weight to each parameter depending on its contribution to the fault. This is done by monitoring changes in parameters during the period the Web service has changed state from responsive to failed service.

This analysis can also be used to estimate the time required for treatment of the failure and to determine the type of faults' persistence (permanent, transient, temporary, or intermittent) [1].

In order to facilitate early detection and prediction of faults, we have set a margin on Web service health score. We consider Web service states below this margin as critical intervals. Doing so provides the self-healing mechanism with enough time to prevent the occurrence of faults or degradation of quality of service. With knowledge of the critical interval, Web services will always be in a range of healthy state. This enables the Web service to work smoothly by avoiding conditions that lead to critical states.

We will use data that are gathered in the Knowledge Acquisition phase as a basis for calculating HS, which will be used for decision making during the Execution phase without requiring the Web service to calculate and return TTL that would exert an additional load on the Web service. In order to get TTL from sources other than the Web service itself, we need to utilize performance parameters from the system and find a relationship between them and TTL. Since determination of such a relationship was not accurate enough through decision trees or rule mining due to the complexity of the relationships and the weights of different parameters, we trained an artificial neural network to predict the value of HS based on normalized values of monitored parameters.

The artificial neural network used in this study was a multi-layer Perceptron composed of 4 layers, 2 hidden layers, and 2 I/O layers. The input layer had 12 processing elements, first and second hidden layers had 10 and 5 processing elements with hyperbolic tangent, respectively, and the output layer had 1 processing element.

The output of the Knowledge Acquisition phase was a trained artificial neural network capable of predicting Health Scores based on normalized values of monitored parameters in the Execution phase.

3.2. Execution Phase

During this phase, we use the same tool that was used for data logging to monitor Web services and their working environment. As mentioned previously, this tool imposes little overload on the system and it is suitable for doing quick analysis of current state of a Web service and its working environment. Data gathered by the monitoring tool are sent to the normalization tool to apply normalization coefficients to each parameter; normalized values are then sent to the analysis tool, the artificial neural network that was previously trained during Knowledge Acquisition phase. The analysis tool consequently calculates the value of HS based on normalized parameters.

Once HS is determined, a tool for analysing HS and predicting the probability failure and the possible time of failure occurrence is required. There are several ways for doing this task, one of which is the curve fitting, e.g., five-point curve fitting wherein the five most recent Web service scores alongside its log times are fitted to a connecting curve and the time of intersection (i.e., t_{HSc}) with critical Health Score limit (i.e., HS_C) that system administrators set is calculated. If t_{HSc} is less than the permissible value set by system administrators, Web service is considered at risk and self-healing strategies are sought. Figure 2 shows an example of the five most current Web service scores fitted on a curve.

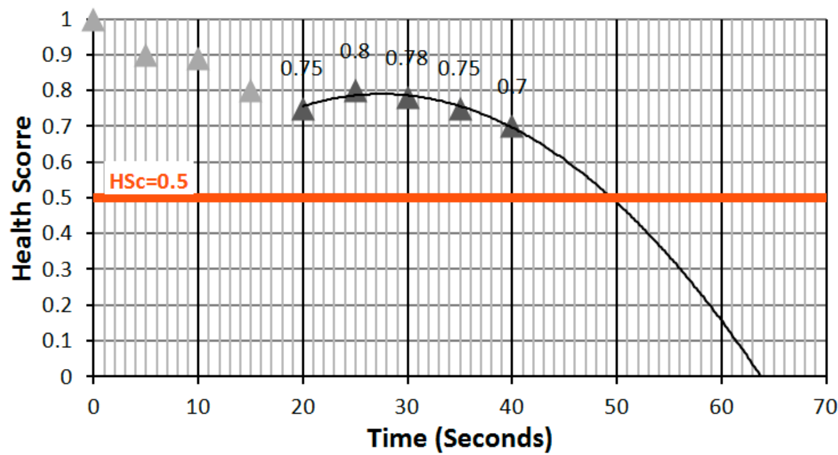

Figure 2: Example of five-point HS curve fitting

If the intersection of the fitted curve and HS_C is close, detailed evaluation of Web service and parameters of the working environment is undertaken. This helps determining which parameter changes are more likely to push the system to a critical state. This is achieved through comparison of correlations between all normalized parameters and HS. If the predicted time of intersection with critical HS line is higher than the maximum range that is set by the system administrators, it can be

concluded that self-healing has been successful and it can be stopped in order to lower operational load on the system, and continue with normal monitoring of the Web service and its working environment.

Figure 3 shows the execution flow of our proposed mechanism.

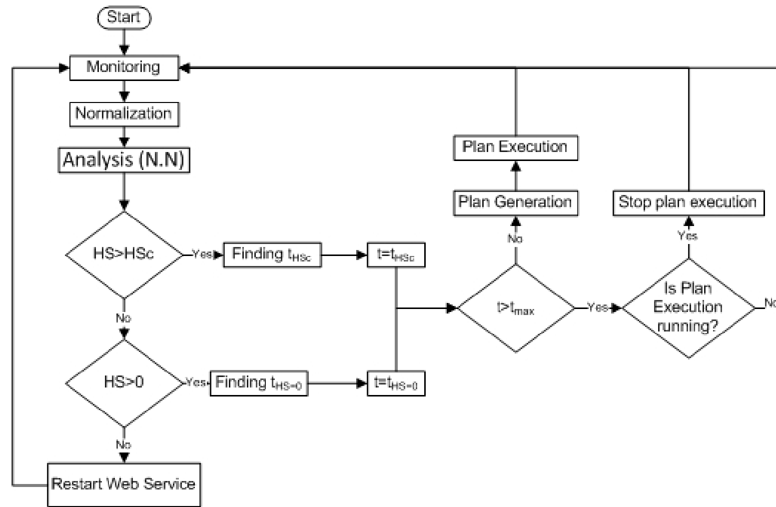

Figure 3: Flowchart of decision making based on HS and time (t)

In our proposed mechanism, we have used a database, known as Rules Database, for storing rules that determine necessary actions for critical states, along with their attributes, including name, usage conditions (appropriate range of usage), and execution priority.

Comparing the state of a Web service and the parameters of its working environment to usage conditions of rules, the Plan Generator tool selects one or more rules from the Rules Database and generates a suitable plan in the form of an XML file. As an example, let us assume that the critical health score, HS_C, is set to 0.5 and t_{max} is 300 seconds. Based on the input parameters, predicted health scores from artificial neural network show a continuous decrease. For example, the health score of the system will get down to 0.25 that is the value set for the critical health score in the next 120 seconds, and it will get down to 0 in 200 seconds that is less than value set for t_{max}. This would trigger the Plan Generation routine to come into action. The Plan Generator will analyze the current state of the system from several different aspects such as responsiveness of Web service, RAM usage by the Web service, available RAM, network I/O and other system and Web service specific parameters listed in Table 4. If the criterion for each rule from Table 4 is met, the rule will be sorted based on its priority and will be added to the plan. Once a plan is generated and submitted for execution, it will not be modified and any addition to current remediation is achieved through generation of a new plan. This means that more than one plan may be executed in parallel. The execution priority of the generated plan in comparison with other possible plans that are already under execution is set and the plan is sent to the Plan Execution tool.

The Plan Execution tool takes over the execution of generated plans based on their priority. This allows faster execution of the most important plan in critical states. It also allows cancelling low importance active plans when the Web service health is recovered or when the execution of low

importance plans impose more load on the working environment than expected resulting in degradation of HS.

Table 4 shows a sample rule set used in this study. More detailed explanation about the tasks to be executed is provided in Table 4. There is not a direct link between artificial neural network and the rules; rather artificial neural network measures the health score of the system based on performance measures. If the forecast of health score shows a possible failure in near future, suitable plans are generated and executed accordingly.

Table 4: Sample rule set

Rule Name	Execution Priority	Usage Conditions	Description	Task to be Executed
Rule1	3	available virtual memory < 40% total	low on virtual memory	increase virtual memory
Rule2	2	available RAM < 40% total	low on available RAM	free up RAM
Rule3	3	available HDD < 30% total	low on available HDD	delete unneeded files
Rule4	4	RAM usage by Web service > 40% of total RAM	high percentage usage of RAM by Web service	run garbage collection
Rule5	3	CPU usage by Web service > 70% for over 2 minutes	high percentage usage of CPU by Web service in a prolonged period	check if service is responsive in a period of 2 minutes. If responsive, take no action
Rule6	1	Web service is not responsive	Web service is not responsive	restart Web service
Rule7	2	CPU usage > 90% for over 2 minutes	high percentage usage of CPU while Web service has low percentage usage of CPU	locate CPU consuming service and restart it if not critical, otherwise wait 2 more minutes. If repeated, restart machine

4 Implementation

In order to implement our proposed mechanism, a software package containing the aforementioned tools has been developed that carries out data gathering, data storage, interpretation and analysis, prediction of degradation of the quality of service of a Web service, prediction of faults, planning, and execution of the proposed plan. The software package has been successfully running on a system hosting a commercial Web service for over 18 months. Figure 4 shows a screen capture of the software package that had been developed for this purpose.

To achieve reliable results in real, we ran the software for one month in training mode with more than 40,000 samples of selected parameters including healthy and faulty states. Most of the samples were collected during peak operation time of the Web service, where Web service had more than 3000 requests per minute, which were handled parallel to each other. The large sampling pool played a crucial role in lowering errors during the assignment of weights to different parameters and the calculation of Health Score. The sampling period was set to 2 seconds during the Knowledge Acquisition phase, and it was set to 5 seconds for the Execution Phase, allowing the system to work smoothly and without much excessive workload. The mean square error, which is the average of the square of the difference between the observed health score and the artificial neural network output,

was 0.000690 for the training dataset that shows a very well trained artificial neural network. Figure 5 shows the cross check between the artificial neural network outputs and health scores stored in the database.

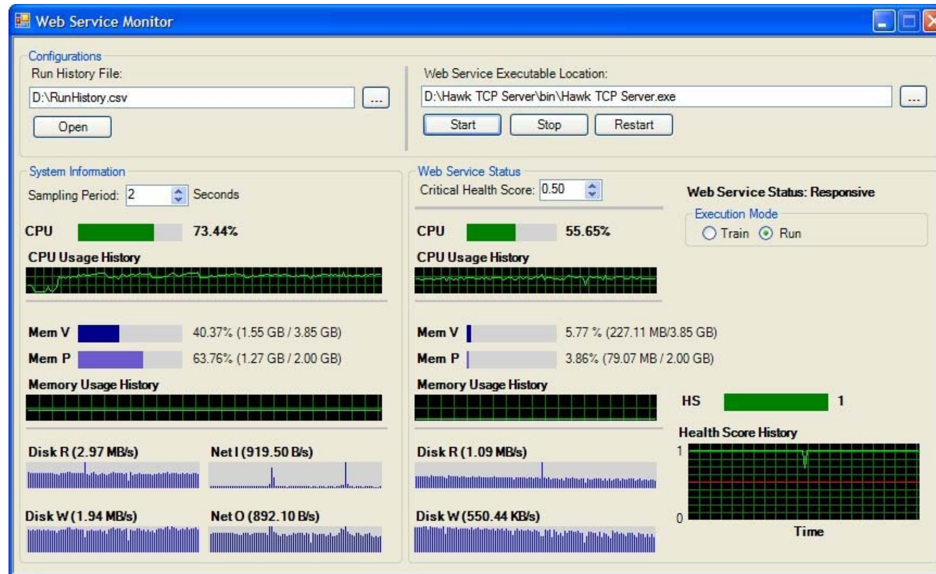

Figure 4: Web service monitor screen capture

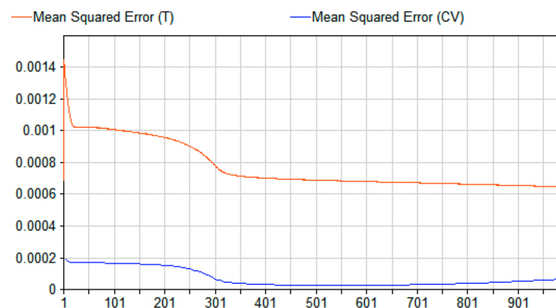

Figure 5: Mean square error of NN training and cross validation

As it is shown in Figure 6, since the outputs from the artificial neural network contain little errors and the predicted health scores match the observed values, the artificial neural network is reliable for the calculation of HS, instead of imposing any load on the system for such calculations. This means that during the Run mode, selection of rules and generation of plans will be based on the health score predicted by the artificial neural network.

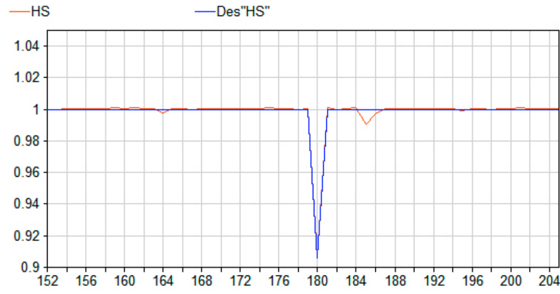

Figure 6: NN output of HS vs. actual HS

5 Discussion

5.1. Evaluation Factors

Since the most important factors relating to the self-healing of Web services are quality of service assurance and high availability of Web services, we consider four factors for qualities for the purpose of comparison:

Unavailability percentage of a Web service, which is equal to

$$UnAvail\ \% = \frac{N}{N_T} \times 100$$

where N is the number of non-responsive states and NT is the total number of recorded states.

Average response time, which is equal to

$$TTL_Ave. = \frac{\sum_{i=1}^{n} TTL_i}{n}$$

where n is the total number of TTLs.

High quality percentage, which is equal to is

$$H.QoS\ \% = \frac{N}{N_T} \times 100$$

where N is the number of states such that $HS_C < HS \leq 1$ and NT is the total number of recorded states.

Low quality percentage, which is equal to

$$L.QoS\ \% = \frac{N}{N_T} \times 100$$

where N is the number of states such that $0.25 \leq HS \leq HS_C$ and NT is the total number of recorded states.

5.2. Evaluation

Table 3 shows the comparison between the evaluation factors before running our proposed self-healing mechanism on the target Web service and the hosting machine, with those factors of Web service and hosting machine after running the proposed self-healing mechanism on the system.

As it is shown in Table 5, under our proposed mechanism, unavailability was reduced by about 70%, average TTL was reduced to 35.2 ms from 43.82 ms (i.e., 20% improvement), high quality of service states was increased by 13 percent, and low quality service occurrence was reduced by 13 percent. Table 6 shows the low overload imposed by our self-healing mechanism on the system.

Table 5: Evaluation factors compared

UnAvail %	TTL_Ave (ms)	H.QoS %	L.QoS %	Situation
0.01	43.82	83	16.99	Without self-healing mechanism
0.003	35.2	96	3.997	Our self-healing mechanism applied

Table 6: Overload imposed to system by the self-healing mechanism

CPU Usage	RAM Usage	HDD Usage
0.5 to 3 %	20 Mega Bytes	50 Mega Bytes

6 Conclusion

In this paper, we presented a new mechanism for self-healing of Web services that could be used as a baseline for the development of new self-healing Web services, as well as self-healing of existing Web services. The proposed mechanism included an added learning phase as compared to the usual self-healing frameworks (MAPE loop).

The concept of Time-To-Leave (TTL) and Health Score (HS) based on TTL and response time was introduced, with system healthiness being expanded from functional/not functional to the broader range of healthy, nearly healthy, intermittent, nearly unhealthy, and unhealthy.

The proposed mechanism was capable of predicting the occurrences of failures using knowledge acquired from Web services and the Web server. Considerable improvements in terms of quality of service assurance and high availability were achieved after application of this mechanism to a commercial Web service under heavy workload. Test results showed that the probability of out of service occurrence decreased by 70% and that the quality of service increased by 13%. The excess workload imposed on the system by our proposed predictive self-healing mechanism was about 3%, whilst the average Time-To-Leave for a request from the client side was decreased by 20%. Due to the preventive nature of this mechanism, delayed responses were not the case after implementation of this mechanism on Web service any more.

Artificial neural network was employed for data analysis. Similar methods, such as data mining, fuzzy logic, and automata, can be studied and used for data analysis as well. Also the prediction tool can be modified to employ a faster or more reliable algorithm for prediction tool, instead of curve fitting. Since our study was conducted on self-healing of Web services, the interaction of Web services can also be added to our studies reported in this paper.

References

1. Pernici, B., Rosati, A. M., Automatic Learning of Repair Strategies for Web Services. In Proceedings of the 5th European Conference on Web Services, Germany, 2007, 119-128.
2. Halima, R. B., Darira, KH., Jmaiel, M., A QoS-Driven Reconfiguration Management System Extending Web Services with Self-Healing Properties. In Proceedings of the16th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, France, 2007, 339-344.
3. Chan, K. S. M., Bishop, J., The design of a self-healing composition cycle for Web services. ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, Vancouver, Canada, May 2009, 20-27.
4. Mostefaoui, G. K., Maamar, Z., On Modelling and Developing Self-Healing Web Services Using Aspects. 2nd IEEE Int. Conference, Bangalore, 2007, 1-8.
5. Naccache, H., Gannod, G. C., A Self-Healing Framework for Web Services. IEEE International Conference on Web Services (ICWS'07), Utah, USA, 2007.
6. Horn, P., Autonomic Computing: IBM's Perspective on the State of Information Technology. IBM Corporation, 2001, 1-39.
7. Kreger H., Web Services Conceptual Architecture (WCSA 1.0). IBM Software Group, May 2001.
8. Kephart, J. O., Chess, D. M., The Vision of Autonomic Computing. IEEE Computer, January 2003.
9. Friese, Th., Muller, J., Freisleben, B., Self-healing Execution of Business Processes Based on a Peer-to-Peer Service Architecture. In Proceedings of the 18th International Conference on Architecture of Computing Systems (ARCS '05) in Systems Aspects in Organic and Pervasive Computing, 2005, 108-123.
10. Papazoglou, M. P., Heuvel, W. J., Web Services Management: A Survey, IEEE Internet Computing. December 2005, Volume 9, Number 6, 58-64.
11. Fensel, D., Bussler, C., The Web Service Modelling Framework WSMF. Electronic Commerce Research and Applications, 2002, 113-137.
12. Naraynan, S., Mcllraith, Sh., Simulation, Verification, and Automated Composition of Web Services. In Proceedings of the 11th International Conference of WWW, Honolulu, Hawaii, ACM Press, 2002, 77-88.
13. McIlraith, Sh., Son, T., Adapting Golog for Composition of Semantic Web Services. In Proceedings of the 8th International Conference of Knowledge Rep. and Reasoning, 2002, 482-493.
14. Zheng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q. Z., Quality-Driven Web Services Composition. In Proceeding of International WWW Conference, ACM Press, 2003, 411-421.
15. Drira, Kh., Molina, M., Nabuco, O., Peralta, L. M. R., Villemur, Th., Product Data and Workflow Management. In Proceedings of Cooperative Environment for Distributed Systems Engineering, LNCS, 2001, 107-151.
16. Deora, V., Shao, J., Shercliff, G., Stockreisser, P.J., Gray, W.A., and Fiddian, N.J. Incorporating QoS Specifications in Service Discovery. In Proceedings of WISE Workshops. 2004, 252-263.
17. Gurguis, Sh., Zeid, A., Towards Autonomic Web Services: Achieving Self-Healing using Web Services. SIGSOFT Softw. Eng. Notes, Volume 30, Issue 4, July 2005, 1-5.
18. White, S. R., Hanson, J. E., Whalley, I., Chess, D. M., Kephart, J. O., An Architectural Approach to Autonomic Computing. In Proceedings of the International Conference of Autonomic Computing, May 2004, IEEE Press, 2-9.
19. Baresi, L., Ghezzi, C., Guinea, S., Towards Self-healing Composition of Services, Contributions to Ubiquitous Computing. Studies in Computational Intelligence, Volume 42, Springer 2007, 27-46.
20. Fuad, M. M., Oudshoorn, M. J., Transformation of Existing Programs into Autonomic and Self-healing Entities. In Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS '07), 2007, 133-144 .

21. Park, J., Yoo, G., Lee, E., Proactive Self-Healing System based on Multi-Agent Technologies. In Proceedings of the 3rd ACIS International Conference on Software Engineering Research, Management and Applications, 2005, 256-263.
22. Han, X., Shi, Z., Niu, We., Lin, F., Zhang, D., An Approach for Diagnosing Unexpected Faults in Web Service Flows. In Proceedings of the 8th International Conference on Grid and Cooperative Computing (GCC '09), 2009, 61-66.
23. Anghel, I., Cioara, T., Salomie, I., Dinsoreanu, M., Rarau, A., A Policy Driven Self-healing Algorithm for Context-Aware Systems. In Proceedings of IEEE 5th International Conference on Intelligent Computer Communication and Processing, 2009, 229 - 236.
24. Noui-Mehidi, A., Self-Diagnosis and Self-Regulation through Performance Monitoring and Tuning. In Proceedings of World Conference on Services-I, 2009, 235 – 242.
25. Alonso, J., Torres, J., Gavalda, R., Predicting Web Server Crashes: A Case Study in Comparing Prediction Algorithms. In Proceedings of the 5th International Conference on Autonomic and Autonomous Systems, 2009, 264 - 269.
26. Yan, P. Y., Pencole, Y., Cordier, M.O., Grastien, A., Monitoring Web Service Networks in a Model-based Approach. In Proceedings of the 3rd IEEE European Conference on Web Services, Vaxjo, Sweden, 2005, 14-16
27. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C., Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing, Volume 1, Issue 1, 2004, 11-33.
28. Candea, G., The Basics of Dependability. CS444a Course Material, Stanford University, 2003.