Software - Implemented Self-healing System

Goutam Kumar Saha

Centre for Development of Advanced Computing, Kolkata, India.
Mail to: CA – 2 / 4 B, Baguiati, Deshbandhu Nagar, Kolkata - 700059, West Bengal, India. gsaha@acm.org

Abstract

The term "Self-healing" denotes the capability of a software system in dealing with bugs. Fault tolerance for dependable computing is to provide the specified service through rigorous design whereas self-healing is meant for run-time issues. The paper describes various issues on designing a self-healing software application system that relies on the on-the-fly error detection and repair of web application or service agent code and data. Self-Healing is a very new area of research that deals with fault tolerance for dynamic systems. Self-healing deals with imprecise specification, uncontrolled environment and reconfiguration of system according to its dynamics. Software, which is capable of detecting and reacting to its malfunctions, is called self-healing software. Such software system has the ability to examine its failures and to take appropriate corrections. Self-Healing system must have knowledge about its expected behavior in order to examine whether its actual behavior deviates from its expected behavior in relation of the environment. A fault-model of Self-Healing system is to state what faults or injuries to be self-healed including fault duration, fault source such as, operational errors, defective system requirements or implementation errors etc. Self-healing categories of aspects include fault-model or fault hypothesis, System-response, System-completeness and Design-context. Based on many important literatures, this paper aims also to illustrate critical points of the emergent research topic of Self – Healing Software System.

Keywords: Self-Healing, Error Detection, Error Repair, Operational errors, Implementation issues, Application semantics, Run time issues.

1. INTRODUCTION

The Self-healing term is used to mean to recover from run-time failures. A system needs to be able to notice an error or injury and then to act on it. In the field of software, an error or injury is identified usually as a failure of a participating machine. A failure can happen in its hardware or the software running on it or a malfunctioning communication connection. It is expected to repair itself in a logical way. Self-healing system needs to be able to discover, diagnose and try to react to bugs and failures. Self-healing components can detect system malfunctions and initiate policy-based corrective action without disrupting the environment. Corrective action may involve a product altering its own state or effecting changes in other components in the environment. The remaining part should try to find a way to continue work without the faulty part. Although there are many possible crashes in different systems, three normal problems are defective hardware, crashed software and broken connection. One of the monitoring mechanisms is using broadcasting. Each agent periodically broadcasts a message of aliveness to its neighbors, and keeps track of when it last heard a reply from its neighbors. If a neighbor is missing, the agent can trigger some response. This primitive mechanism contains a tradeoff. The frequency of aliveness messages from neighbors

determines the speed with which faults can be detected and the lag in response time. Faster response time comes with a higher cost of communication. Crashes management can be carried out in many ways. If a site crashes, all data on this site is irrevocably lost. Therefore, we need this data to be copied in advance, so that a redundant copy can be fetched and used to continue work. An obvious solution is to duplicate all data, so that in the case of a crash the lost data still has a copy anyway. However, to accomplish this, a copy of all data must be sent immediately over the network, much communication overhead can be expected. Second approach is to keep all applied data in memory until they are executed and have on their part applied new data to others. So in case of a data loss, the missing executions could be redone. This would require a powerful and costly mechanism to decide which data can be deleted. System-response includes the aspects of fault detection, degree of degradation, fault response and an attempt to recovery action or compensation for a fault. Fault detection approaches involved in a self-healing system include application system's semantics-driven assertions [13, 14], supervisory checks, examining the computing answers, comparison of replicated components, online self testing etc. System-completeness aspect deals with reality of knowledge limits, incompleteness in specifications and designs thereof. It also deals with the problem of system self-knowledge, system evolution etc. Handling the architectural incompleteness for example, of third-party components or of various patches during or after system deployment is really a challenging issue in developing a self-healing system. A self-healing approach is to run-time validate the various assertions that are derived from application semantics for designing a Self-healing system. The assertions are derived from the application domain specific knowledge of the system's expected behavior and Self-Healing [1,2,3,4,5,6,7,8,9,10,11,12] achieved by examining whether the system's actual behavior deviates from its expected behavior in relation of the system's environment.

The proposed self-healing approach, as described in section 3.1, aims to detect errors on-the-fly in web application's or service agent's code and data and to automatically recover [17] them allowing a system to keep on computing after it has sustained an error that would normally require a restart.

2. SELF-HEALING SOFTWARE ARCHITECTURAL REQUIREMENTS

A Self-healing system has the ability to modify its own behavior in response to changes in its environment, such as resource variability, changing user needs, mobility, and system errors. A self-healing system's lifecycle has the following four major activities as described in [16]:-

(a) Monitoring the system at runtime, (b) Planning the changes, (c) Deploying the change descriptions, and (d) Enacting the changes.

Depending on the above mentioned activities, characteristics of existing architectural [7] issues, self-healing systems have the following architectural requirements:-

(a) *Awareness:* Supporting the monitoring of the system's performance (state, behavior, correctness, reliability, and so forth) and recognition of anomalies in that performance.

(b) Adaptability: Enabling modification of a system's static (structural and topological), dynamic (behavioral and interaction) and run-time aspects.

(c) Dynamicity: Encapsulating system adaptability during run-time (communication integrity and internal state consistency).

(d) *Autonomy:* Providing ability to address the anomalies as discovered through awareness and observability in the performance of a resulting system and / or its execution environment. Planning, deploying, and enacting the necessary changes can achieve autonomy.

(e) *Observability:* Enabling the monitoring of a resulting self-healing system's execution environment. It is to be noted that the system might not be able to influence changes in its environment (say, re-establishing the failed network links), but it might plan changes within itself in response to the environment (say, performing in degraded mode until the network-link is reestablished).

(f) *Robustness:* Providing ability for a resulting system to effectively respond to unforeseen operating conditions those might be imposed by the system's external environment (such as, malicious attacks, unpredictable behavior of the system's runtime substrate, unintended system usage), as well as errors, faults, and failures within the system itself. This is to be noted that this definition of robustness subsumes fault-tolerance.

(g) *Distributability:* Supporting effective performance of a resulting system in the face of different distribution or deployment profiles.

(h) *Mobility:* Providing the ability to dynamically change the (physical or logical) locations of a system's constituent elements.
(i) Traceability: Relating a system's architectural elements to the system's execution-level modules in order to enable change enactment in support of the above mentioned requirements.

3. SELF-HEALING SOFTWARE SYSTEM'S FAULTS & FIXES

In a typical multi-tier computing infrastructures [11] consisting of web applications and web services, we expect similar ubiquity and reliability from such services as that offered by the electricity grid and phone system. Recent studies [1,2,3,4,5,6,7,8,9,10,11,12] show that 72% of the top-40 web sites suffer user-visible failures, for example, items not being added to shopping carts or various error messages. Such failures occur because of various failures such as Java exception, deadlock threads, aging, source code bug etc. Some of the similar faults and their remedies or fixes [11] as observed in multi-tier J2EE services are stated in table 1. The self-healing part of web applications must deal with failures. The failures can be in both of system level and logical level. System failure can be the failure of say, medicine supplier's web services [5,15]. Logical failure say, can be the medicine supplier does not fulfill the order. After detecting these failures, the web services must be able to make an optimal choice and solve the problem by reconfiguration. However the important principle is the healing behavior must still obey to the policies we specified. For instance, if we set the policy as long-term relationship the solution would be to cancel the order instead of replacing the supplier. Healing mechanism must be created based on combinations of compensation, recovery and an optimization model for making an appropriate plan for healing. So, for designing a self-healing software system, we need to take care of application semantics also apart from other existing error or fault detection techniques. Based on the application's semantics, we need to produce assertions and to validate them during run-time against known set of test data too.

Table 1: Self-Healing System's Faults & Fixes

Faults	Fixes
Source Code Bugs	Reboot Tier / Service and Notification to Administrator
Java Exceptions	Micro-boot EJB
Deadlock Threads	Micro-boot EJB, Kill hung Query
Buffer Contention	Repartition the memory across various Buffers
Aging	Rebooting for reclaiming the leaked resources
Read / Write Contention on Table Block	Repartition Table to Balance Accesses around Partitions

3.1 The Algorithmic Approach of On-the-fly Automatic_Error_Detection_Repair to Self-Healing Web Application Software

The following steps or pseudo-code describe how this on-the-fly Automatic_Error_Detection_Repair algorithm verifies for the presence of multiple data - errors in the web application or service agent code and data, and how it corrects the erroneous data of the contaminated agent code, by comparing an web application's state (using three copies of an web application service agent code and data that are injected at a host upon its arrival) byte by byte. The symbols "/* */" are to enclose remarks only.

/* There are three images or replicas of an agent that are injected at a host. Byte-wise comparisons and corrections are carried out (using XOR) till the last byte of the copies. It verifies the corresponding three bytes at an offset say, *d*, of the three copies of the service agent code and if any byte error has occurred, then it repairs the corrupted byte. Starting addresses of the three images are known. The notation C^{1}_{d} denotes the *d* th byte (at an offset say, *d*) in the agent's first copy or Copy –1. */

Step 1. Set B = Size of an image in bytes. /* Size of an image in bytes is known*/

Step 2. Set $d = 0$ /* initialize the memory offset say, *d**/

Step 3. $R_{12} = C^{1}_{d}$ *.XOR.* C^{2}_{d} /*bytes at offset *d* in images C^{1} , C^{2} are -

Step 4. If R_{12} .EQ. 0 , Then: - XORed and result is stored at R_{12} */

No Error. /* goto step 5 i.e., program control goes out of the outer EndIf of step-4, for scanning the next byte of the three copies */

Else:

$R_{13} = C^{1}_{d}$ *.XOR.* C^{3}_{d}

If R_{13} *.EQ.* 0 , Then:

$C^{2}_{d} = C^{2}_{d}$ *.XOR.* R_{12} /* Bytes at C^{1}_{d} , C^{3}_{d} are correct but C^{2}_{d} is bad, so the erroneous byte at C^{2}_{d} is repaired*/

Else:

$R_{23} = C^{2}_{d}$ *.XOR.* C^{3}_{d}

If R_{23} *.EQ. 0*, Then:

$C^{1}_{d} = C^{1}_{d}$ *.XOR.* R_{12}

/* Bytes at C^{2}_{d} , C^{3}_{d} are correct but C^{1}_{d} is bad, so the erroneous byte at C^{1}_{d} is repaired*/

Else:

Call *HARD_ERR*

/*All the three bytes at the same offset *d* in the three copies are corrupted i.e. all the three copies are corrupted - indicates a memory device problem or permanent errors, Call *HARD_ERR* routine to restart the agent execution. */

{End of If structure}

{End of If Structure }

{End of If Structure }

Step 5. Set $d = d + 1$ /* offset *d* is incremented by one to scan next byte */

If $d < B$, Then: /* Scan the next byte for error detection & repair thereof */

GOTO Step 3.

Else:

Return

/*After the entire scan & repair, starting from 0^{th} byte through (B-1) th byte in the 1^{st}, 2^{nd} and 3^{rd} copies simultaneously, program control goes back to the primary copy of agent and agent execution continues in an application based on service agents */

{End of If structure}

The possibility of getting inadvertently alteration (by transients) of two similar bytes in two copies (located at two distant locations) to mean a some other value resulting in a similar corrupted byte-pattern, is negligibly small. In other words the chances of byte error remaining undetected is:

$$(1 / (2^8)) * (1 / (2^8)) = 1 / (2^{16})$$

The above procedure verifies and recovers byte errors in the entire web application or service agent code and data, by increasing the value of offset *d* from 0 to B–1 (the size of an image is of say, *B* bytes). This is very effective for soft errors (induced during its transit or at reception at a host) detection and corrections of code and data prior to its execution on a host. After detecting and repairing the entire code and data, program control goes to start the execution of an agent's copy. This approach can detect 38% errors (i.e., fault coverage) in a typical fault injection (i.e., causing random bit errors) experiment, whereas hardware Error Detection Mechanisms (EDM) at a host could detect 26% errors. Even an entirely corrupted agent code can be repaired by this effective technique of self-healing that relies on this on-the-fly error detection and repair.

4. CONCLUSION

A useful algorithmic approach to on-the-fly automatic data error detection and recovery using moderate code and time redundancy of the order of 2.6, has been described in this paper. This paper has also described various important issues of designing a self-healing system based on available literatures. In order to develop an effective self-healing system model we must be conversant to all these points as described here. The proposed approach is an important one toward Self – Healing software design, which is a much research topic for developing reliable web services and web applications.

References

[1] Jun, L. *Self-X Property and Application in Web Services.* Fachbreich Informatik ,TU Darmstadt, 2006.

[2] Baresi, L. and Guinea, S. *An Introduction to Self-Healing Web Services,* Dipartimento di Elettronica e Informazione-Politecnico di Milano, 2005.

[3] *A Technical Introduction: Predictive Self-Healing in the Solaris™ 10 Operating System.* September 2004.

[4] Tosi, D. Research Perspectives in Self-Healing Systems. *Technical Report, LTA:2004:06,* University of Milano-Bicocca, 2004.

[5] Sidiroglou, M.E. and Angelos, L. and Keromytis, D. Hardware Support for Self-Healing Software Services. *ACM SIGARCH Computer Architecture News,* Vol. 33, No. 1, 2005.

[6] Sidiroglou, S. and Keromytis, A. D. A Network Worm Vaccine Architecture. *In Proceedings of the IEEE Workshop on Enterprise Technologies: Infrastructure for Collaborative Enterprises (WETICE),* Workshop on Enterprise Security, pp. 220–225, June 2003.

[7] Mikic-Rakic, M. and Mehta, N. and Medvidovic, N. Architectural Style Requirements for Self-Healing Systems. *Technical Report of Computer Science Department,* University of Southern California, Los Angeles, CA 90089-0781, 2002. USA.

[8] Tosi, D. Research Perspectives in Self-Healing Systems. *Technical Report* of the University of Milano-Bieocca, 2004.

[9] Koopman, P. Elements of the Self-Healing System Problem Space. *Proceedings of the ICSE WAD03,* 2003.

[10] Lemos, R. ICSE 2003 WADS Panel: Fault Tolerance and Self-Healing. *Proceedings* of the ICSE 2003.

[11] Cook, B. and Babu, S. and Candea, G. and Duan, S. Toward Self-Healing Multitier Services. *Technical Report* of the Duke University, 2005.

[12] Saha, G.K. Self – Healing Software. *ACM Ubiquity,* Vol. 8, No. 12, 2007.

[13] Saha, G.K. Application Semantic Driven Assertions toward Fault Tolerant Computing. *ACM Ubiquity,* Vol. 7, No. 22, pp. 1 - 27, ACM Press, 2006, USA.

[14] Saha, G.K. Software Fault Tolerance through Run – Time Fault Detection. *ACM Ubiquity,* Vol. 6, No. 46, pp. 1-5, ACM Press, 2005, USA.

[15] Saha, G.K. Fault Tolerance in Web Services. *ACM Ubiquity,* Vol. 7, No. 9, ACM Press, 2006, USA.

[16] Oreizy, P. and Gorlick, M.M. and Taylor, R.N. and Heimbigner, D. and Johnson, G. and Medvidovic, N. and Quilici, A. Rosenblum, D.S. and Wolf A.L An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems,* Vol. 14, No. 3, p.54-62, 1999, USA.

[17] Saha, G.K. Transient Fault Tolerance in Mobile Agent Based Computing. *Infocomp Journal of Computer Science,* Vol. .4, No. 4, pp. 1-11, UFLA Press, 2005, Brazil.