A Case-Based Framework for Self-Healing Paralysed Components in Distributed Software Applications

Conference Paper · December 2014

DOI: 10.13140/2.1.1616.6728

3 authors:

Tanim Hasan

University of Dhaka

1 PUBLICATION **0** CITATIONS

SEE PROFILE

Asif Imran

University of Dhaka

18 PUBLICATIONS **26** CITATIONS

SEE PROFILE

Kazi Sakib

University of Dhaka

40 PUBLICATIONS **52** CITATIONS

SEE PROFILE

A Case-Based Framework for Self-Healing Paralysed Components in Distributed Software Applications

Tanim Hasan, Asif Imran and Kazi Sakib
Institute of Information Technology
University of Dhaka, Dhaka-1000, Bangladesh
Email: bit0301@iit.du.ac.bd, asifimran@du.ac.bd, sakib@iit.du.ac.bd

***Abstract*—Self-healing is the ability of the software to detect faulty modules at execution time and replace or recover those without affecting other components. This paper proposes a framework for self-healing of Distributed Software System (DSS). $Monitoring$ component is used to detect and record failures of DSS. $Healing$ system will replace the paralysed components with healthy ones which will be initiated from the information given by $Monitoring$ system to the proposed $Reviver$ process. A failed case table is required to match the real life failures with it for identification of the solution. Distance between the failed case table and the recorded failures need to be calculated using exclusive OR since the solution closest to the fail can then be determined. Afterwards, the minimum distance between those is used to resolve the failure. This recovery is achieved through replacement of the faulty modules with redundant components in the DSS. Performance evaluation shows a desirable time consumption of less than the standard 0.7 seconds for component replacement in all the experimental iterations.**

Keywords*—Software self-healing, Distributed computing, Software engineering, Complex system management.

I. Introduction

Self-healing is a mechanism that autonomously detects failures and responds to those at run-time without human intervention. Component based self-healing in DSS make the necessary adjustments to restore the application towards normal operation by reviving the depended processes, components or whole system. Software system has become too complex to manage and fix manually. Applications are now designed with so many depended processes and distributed components that it is hard to immediately react to failures manually. Naturally in distributed environments, depended processes may run in different virtual machines of an application. When a process fails, other processes have no idea why application or system gets down. In this case, a system administrator is required to detect failures manually and take initiatives as per demand, leading to increased recovery time and expenses. Early detection of paralysed components at run-time is highly important to sustain the errors and prevent those from propagating to other components of the application [1], [2].

Ensuring fault tolerance in modern distributed software requires the components of those to self-heal, that is to detect causes of component paralysis at the execution time. Redundant components must be used to replace the paralyzed ones together with state restoration in short time and without affecting the healthy ones. The complex nature of distributed applications present significant challenges to ensure self-healing of those. Research needs to be conducted to detect key causes of software paralysis at run-time and replace the failed components of the application with new ones.

978-1-4799-6399-7/14/$31.00

Legacy applications were injected with self-healing primitives to ensure recovery of software modules after failure [3]. Run-time errors were identified as attributes that were recovered through the proposed primitives. The methodology is applicable to codes specific to Java applications. Bytecodes were analyzed to detect run-time issues, and modules suffering from run-time errors were recovered through reinitiation. However, the mechanism proposed by the authors involved termination and restart of the modules from initial state, hence the importance of ensuring state restoration of critical processes were not considered. Security perspective of cloud based applications were taken under consideration and malicious attacks on cloud components were detected in [4]. However, the authors did not consider the importance of redundant components for fast recovery of paralyzed modules in cloud based software.

The proposed mechanism is a three layered framework to detect faults in software components and recover from those in short time. The first layer is called the $Monitor$ that provides a failure case table containing predefined software fault conditions for components together with the solutions identified for those. The $Monitor$ module also analyses real life software components to determine and list failures. A paralysed component is the one that does not perform its target functionality due to certain failures. Next, $Healing$ module of the framework determines the distance between the detected faults and the ones listed in the failure case table using exclusive OR. The minimum distance found in this way will yield the closest solution to the current problem [5]. The entry in the failure case table that has the minimum distance is considered as the solve to the paralysis problem, hence the solution prevalent for that case is applied to the paralyzed component. This failure case table is used to compare with real life cases and determine solution for those. Afterwards, based on the solution the $Reviver$ module replaces the faulty components with redundant ones preserving the last active state. Redundant copies of healthy components are stored in additional cloud virtual machine (vm) instances that are used to replace the paralysed ones.

Empirical analysis of performance yields desirable results based on a case study resembling real life scenario of an e-commerce website. Each component of the website resides in

individual vm-instances of the cloud, hereby ensuring decentralization of services. The proposed framework is appended into the e-commerce software as additional components. Next, faulty iterations of the application were executed and faults collected for those. Upon calculation of exclusive OR between the failed test case and real life cases, minimum distance of detected faults that are determined within the range of 0-2 units which is desirable. Time consumption for replacing paralyzed component with a health component in cloud vm-instance was found to be less than 0.7 seconds in four iterations, resulting in desirable time complexity specified in [5].

II. Related Work

Application based failure detection model has been proposed in [6]. Sensors are implemented into the existing codes that trigger alarms when a failure of code execution occurs. Information about the causes of failure is detected and a solution is suggested to the users from a solution set that is predefined. The suggestions are used by the clients to recover from the failed state. The components are stacked into one global module that helps the analysis of failures in those. However, the increased affinity to failure caused by a single global component has not been considered here. Additionally the importance of using distributed component architecture to ensure uptime of other services when one fails has been considered to a minimum extent.

Rule based methods to detect software faults based on programmer induced failures have been presented in [7]. Prespecified failure conditions have been considered by the authors and fails were detected based on those. Failure attributes like buffer overflows and coding errors were searched for in the code and possible code blocks were identified and suggested for correction. However, the problem due to sudden process termination caused by component failure was not considered.

Self-healing mechanism of critical sections of components has been proposed in [8]. A set of components, connectors and healing parts are installed at each node. At the same time, the components place a copy of the failure condition rules in the description repository of the proposed framework. Policy based analysis have been used to generate dependency graphs. Based on those, the self-healing policy is chosen that involves removing the failed processes and recovering those from initial state. Since the policy enforces the failed processes to start from scratch, the importance of restoring components from the last working state have not been taken under consideration.

Market based heuristics have been used to ensure adaptation of cloud virtual machine (vm) instances to changing scenarios [9]. Self-adaptation has been regarded as a critical Quality of Service (QoS) requirement for the cloud. Continuous double action algorithm has been proposed to allow services to choose from the components available. In addition, how cloud computing minimizes cost through self-healing has been identified in [10]. The choice is based on the resource allocation on various vm-instances, where the vm with resources nearest to satisfying the process requirement is selected and the component is configured in it. However, the self-adaptation mechanism is active during system initiation phase only and does not take into consideration the self-recovery required during system runtime.

Unanticipated changes to the source code that violate assumptions and internal structure of the programs at compile time have been addressed in [11] The system is divided into two parts, firstly the functional part that implements the expected requirements and secondly the self-healing part that defines the policies of automatic failure detection and recovery. The two modules are defined separately and the failure module is used to encapsulate the functional and self-healing components. The fault model defines a pre-determined set of faults. However, the faults defined in the fault model belong to compile time issues of software only, runtime faults have been considered to a minimal extent by the authors.

Mechanism for translating anomalies in software based on pre-defined failure states haven proposed in [12]. The model specifies the characteristics and goals of the system through analysis of the inputs and outputs that are generated after processing those. Next, state rules are created based on the internal and external constraints of the system. On the basis of the defined state rules, a dependency graph is created that is used to detect what components are affected by a failed process. However, monitoring the system to identify faults at run time was not taken under consideration.

III. Component Based Self Healing Mechanism In Distributed System

The rising complication of software systems demands innovative ways of control over the systems. Systems should be able to adapt dynamically to changes in the environment and components. An emerging methodology to overcome this problem is software self-healing. Architecture for component based self-healing mechanism in distributed system is proposed in this section and depicted in Figure 1. If an application goes down due to the process that are running on VM, then system admin needs to be involved to revive the component, which is costly and less efficient. The challenge of the objective is to detect failures and revive them at run-time in distributed system. Proposed framework of self-healing systems are classes of software systems that exhibit the ability to detect failures at run-time and revive the whole system autonomously. Life cycle of component-based software self-healing in distributed system consists of four major activities:

- Monitoring the components and depended process
- Detecting the failures issues
- Notifying the main the application about the defective components
- Reviving the components as per policy

A. Overview

During the design of the architecture, attention is given to component based self-healing as well as fitting the framework in distributed environments. DSS provides the outcomes to users with the help of multiple components while the depended components in different virtual machines are not aware of other components operating properly or not.

The architecture performs the healing action with a $Monitor$ program and transmits the information to the

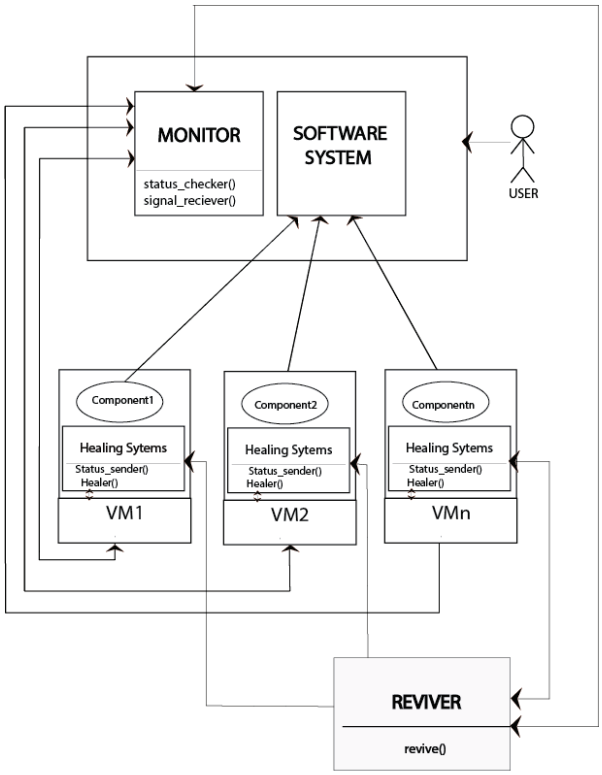

Fig. 1. Communication of the modules in proposed framework

Reviver process. However, the virtual machines have command listener to heal the components as per policy and to send the status to monitor program. The top-level overview of the proposed architecture is shown in Fig 1. The architecture is divided into three segments while the whole component based self-healing for DSS stands on three core tasks that are:

- *Monitor* module
- *Reviver* process
- *Healing* system

The first is *Monitor*, which will run along with main application. *Monitor* is designed such that it will store all the depended components and processes information (e.g. process number, process id, VM information, etc.) by checking regular status. Next, it will send the faulty processes or components that responds to the *Reviver* to the *Healing* system.

The next one is *Reviver*, which will collect the process's or component's information that needs healing from component *Monitor*. It sends the request to the *Healing* system with the faulty process information and restoration data. After healing or recovering that faulty process it will send feedback message to the component monitor. The final segment is *Healing* system, which will respond to *Revivers* request and heal the processes for performing normal operation. All three segments are designed such that it should fit in complex *DSS*.

B. *Monitoring system and error detection*

Component *Monitoring* system will monitor the processes or components. With a failure detection methodology it will identify the failures then send it to the reviver to re-initiation. Anomalies that were considered can happen to software system are process crash in computing, run-time-error of a depended process, communicating connections problem and resource over allocation. The four major failures stated here are identified by *Monitoring* system. Each of the failure and its identification procedure are discussed in this section.

Process crashes when it performs an operation that is not allowed by the Operating System (OS). If a process is attempting to access (read or to write) at a specific memory block which is not allowed for that specific process, the process get crashed which is called segmentation fault. Also while attempting to execute invalid instructions, to perform inaccessible I/O operations and passing invalid arguments to system calls can lead a process to dump [13].

In some critical conditions, process attempts to execute machine instructions with bad arguments, for example divide by zero, function on denormalization or NULL values can make the process crash. Monitoring system will identify this process crashes by identifying the error messages and checking the time duration that has elapsed. If a process does not respond to main application within the threshold time then it will be considered as a paralyzed depended process. Segmentation fault and unauthorized faults will be detected from the error messages. Run-time error occurs during the execution of a program [14]. Such as running out of memory is a run-time error. By setting up a threshold value for each parameter like load time, memory, response time etc. monitoring system can identify the run-time error.

Use of self-healing system and process reviver software system can get rid of the run-time error. Error detection and status checker method of reviver will identify this failures, then it can initiate the paralyzed process once again by healing system. Communicating issues in distributed system is one of the major problems [15]. It can be identified by sending request from monitoring system, if virtual machine does not respond to the request within the threshold time then monitoring system will consider this as communicating connection problem. Resource over allocation can be happened in terms of using over memory, bandwidth, operating the process for long time without conveying any error message [16]. On this occasion again proposed monitoring system will identify those by checking with threshold value.

Above stated failures can happen in Distributed Software (DS), monitoring system can identify those failures and take initiatives to revive the process. Reviver will initiate the process with necessary information and if possible with previous data also. Proposed component monitoring system will consider only the four major failures although there are some other issues like functional problem, logical problem etc. will not be under consideration, because those faults are design problems and internal program errors.

C. *Proposed self-recovery algorithm*

Reviver component of the proposed framework collects all the faulty process information and stores those in a journal

as shown in Algorithm 1. $Reviver$ sends the request to $Healing$ system to heal the process and receives to feedback from $Monitor$. Initially there will be an empty $Proc_{list}$ and $Proc_{info}$ register. $Monitoring$ system will store all the process information (vms ip , hardware address, port, process resource allocation, etc). Since it is initialized at the initiation, next it will push both the $Proc_{id}$ and $Proc_{info}$ into the empty list. To get the $Proc_{info}$ $GetProcessInfo(Proc[i])$ method has been introduced. This method will take $Proc_{id}$ as an argument and return all the information of that particular process including vm's basic information.

Algorithm 1 Algorithm for self-healing component

procedure SELF-HEAL($P_{List}, Proc_{info}$)
  $Proc_{list} \leftarrow []$
  $Proc_{info} \leftarrow []$
  $Proc_{list}$ *and* $Proc_{info}$
  while $i = num_{DependedProc}$ **do**
    $SignalReciever(P_{id}, Proc_{info[i]})$
    If $Proc_{[i]} == IsFaultyStatProc[i]$
    $Proc_{info} \leftarrow GetProcessInfo(Proc[i])$
  end while
  while $i = Proc_{list}$ **do**
    Else **return** Process is not paralyzed
    If ($P_{INFO}[I]$ ==IsFaulty($Proc\_status[i]$)), then
    Feedback = Reviver($Proc_{info}, VM_{info}, Policy$)
    Else **return** Message Process is ok
  end while
  $IsFaulty_{Stat_{Proc[i]}}$
  return Faulty = Error_Message_vm
  If $Status\ and\ Proc_{info} == Faulty$
  Else **return** False
  Proc_list.push(Proc[i])
  $Proc_{info} = Get\_Proc_{info(process[i])}$
  $feedback$ = Reviver($stringProc_{info}, vm_{info}$,
  $Policy$)
  return true
end procedure

Although this method call will request vm's, those will respond to the request by collecting information from components including process information. $SignalReciever()$ is another method, which takes $Process_{id}$ and $Proc_{info}$ as arguments. It will return the status of that process or components of the vm. If the return value matches faulty codes then a reviver will send the process with the information for healing. In this process all the journals will be stored as a feedback in the $Monitoring$ system.

If the process is operating correctly it will generate a positive message and notify the administrator. $IsFaulty()$ is a method that will verify the response from vm whether the process performs as desired. The fault detection will be done based on the threshold value set for the system and error messages from the OS.

D. Procedure of Healing system

The proposed framework revive or recover faulty processes using this healing system. The mechanism is provided with the necessary information given by the $Reviver$. It needs the depicted information to heal the process as:

- faulty process information
- previous restoration data
- VM's information

$Healing$ will send the feedback message to the $Reviver$ if it can complete the action. It will store the most recent three healthy components to the vm which had faulty ones. In the case of replacing the components, difference between the stored-set and detected failed-set is determined using exclusive OR between the two. Redundant copies of healthy components are stored in that vm which had faulty components. The stored-set that has the minimum distance is considered as the solve to the paralysis problem, hence the solution is prevalent. The solve included preserving the last working state of the faulty component, replacing it with a healthy one, and restoring the state into the new module. Afterwards the new component is integrated into the software architecture and begins functioning in co-ordination with the other active components.

IV. CASE STUDY

The proposed methodology can be tested through concrete experimentation of the self-healing framework depicted in Figure 2. The realization of the performance can be conducted through implementation of the proposed algorithms in a real life case. The test environment should consist of a web based electronic commerce (e-commerce) site implemented in cloud.

The system consists of a user-friendly web page where customers can register, post adverts of the product, browse new products and book those for purchase. Hence a web-service component is required for the operation of the e-commerce website. In this case Nginx web server is used coupled with a PostgreSQL database that is configured in a separate vm-instance. The PostgreSQL database server stores and accepts queries for all user registrations, posting adverts and booking items through the application web service [17].

A. Integration of Monitor Module

Upon receiving a failure condition cause due to external queries mainly from custom searches, the Monitor module detects the issue, sorts the problem and assigns it to a specific problem group that is recorded in the data storage. Based on the rules set against each problem the Monitor identifies the faulty component running in the paralyzed vm instance, records the last working state and dependencies of the component using recovery orientation. The Monitor algorithm is implemented in $Python$ and requires $Twisted$ to execute in distributed environments.

Recovery procedures include isolation of the paralyzed component. Next a new vm from the redundant array of vms is allocated and the state of the paralyzed vm is restored in it. The new vm becomes functional when it is integrated with the remaining vms running the components of the e-commerce web site [18]. Hence, the issue of failed components is self-healed using the redundant array of vm-instances in the cloud environment. The cloud framework that can be used for experimentation is OpenStack Icehouse running on CentOS 6.5 servers.

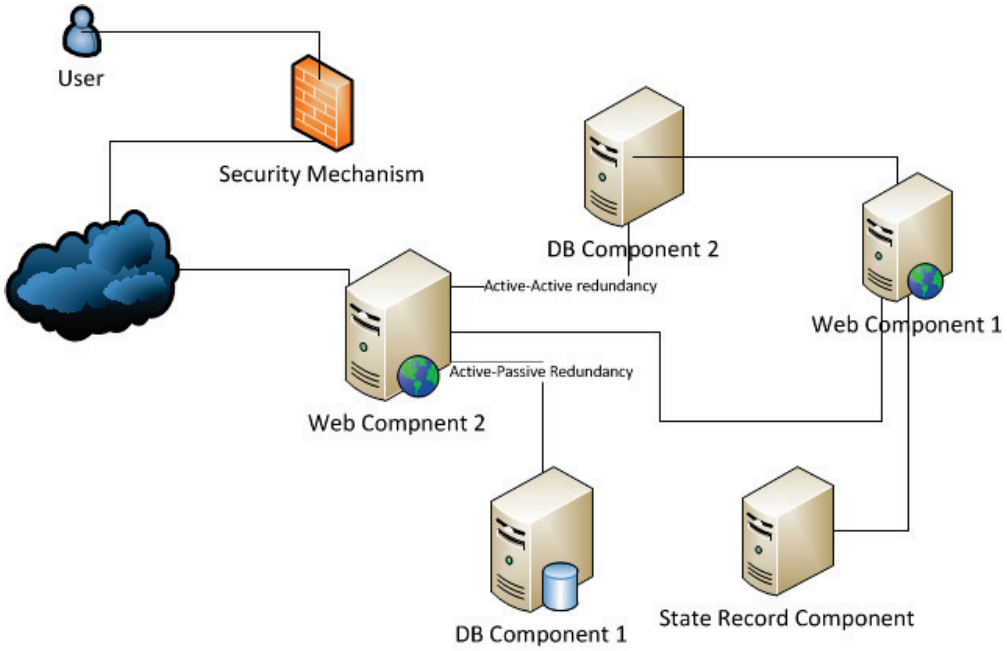

Fig. 2. Self healing components with redundancy for e-commerce application

TABLE I. DECLARATION OF PREDEFINED FAILURE CASES IN DISTRIBUTED SOFTWARE

Failure Cases	NginxConf	NginxComm	PermitErr	RegistrationErr	PostgreSQLConf	Solution
User_Registration	Yes	No	Yes	Yes	No	Restart Nginx, allocate vm and set permission
Post_Advert	No	Yes	Yes	No	Yes	Allocate new vm and reload states
Browse_Page	No	Yes	No	No	Yes	Trigger configuration upon restart of Nginx
Book_Item	No	No	Yes	No	Yes	Allocate new vm and restart PostgreSQL
Visit_Home	Yes	No	No	No	Yes	Allocate new vm and load Nginx state

TABLE II. MINIMUM DISTANCE OF TEST ITERATION FROM PREDEFINED CASES

Failure Cases	NginxConf	NginxComm	PermitErr	RegistrationErr	PostgreSQLConf	Minimum distance
Iteration-1	1	0	0	0	0	1
Iteration-2	1	0	1	0	1	1
Iteration-3	0	0	1	0	0	2
Iteration-4	1	1	0	1	0	1

B. Two layer distributed component

At the application layer, individual components are configured in separate vm-instances of the cloud. Hence distributed component architecture is achieved where separate services are designations to separate vms. The e-commerce website is hosted in the cloud using a 2-layer implementation mechanism [19]. The top layer consists of vm containing the web server and pages. Hence the Domain Name Service (DNS) component is also configured in the vm-instance to ensure separation of services. The first tier consists of the vm that contains web pages with which customers can communicate and search for desired goods in the e-commerce website. The operations at the client end involves the component running at this vm-instance that include registering, posting new adverts and selling products.

The second vm consists of PostgreSQL database that stores data entered by users. Two tables are added primarily to the database namely $UserInformation$, $ItemList$ and $Price$.

All the components are implemented in cloud datacenter running OpenStack Icehouse on CentOS 6.5. The vm-instances are allocated Random Access Memory (RAM) of 1 GigaByte (GB) each and 4 virtual Central Processing Units (vCPU). The application tier has 10 GB of disk drives and the DB tier has 30 GB of hard disk storage allocate using the cloud.

C. Interaction of distributed components

The paper takes into consideration that the wide array of operations like registration of new users, searching for products, adding new products for sale, reporting items, etc are conducted by the e-commerce website under consideration in the case study. Hence the results that can be obtained from the experimental testbed are applicable to the entire class of standard e-commerce websites.

The proposed $Monitor$ plugin is developed and implemented in the e-commerce components to track and diagnose failure of those. Algorithm 1 identifies the $Monitor$ mechanism and states the inputs, observations and notification measures. More specifically, the service tracked by $Monitor$ are summarized as: $User\_Registration$, $Post\_Advert$, $Browse\_Page$, $Book\_Item$, and $Visit\_Home$. Under the given scenario, the following failures are identified to occur frequently in software components.

- **Web-server failure:** The request overflow problem can occur in the web server.
- **Server to Application Communication failure:** This isse occurs when the webserver fails to communicate with the application component due to network service termination. In case of Service Oriented Architecture (SOA) this issue may occur when one vm component is manually shut down and the other vm's cannot communicate with it.
- **PostgreSQL failure:** PostgreSQL configuration issues may result in failure of the database component, which occurs during incorrect configuration of the DB vm.
- **File permission failure:** This failure case occurs when the users do not have permission to execute database write operations in the DB component of the e-commerce website.

The User_Registration activity will require the component to access he $UserList$ table in the $DatabaseBase$ (DB) component, whereas Browse_Page require access to the $ItemList$ table, $Visit\_Home$ is a simple command to load the home page of the website and $Book\_Item$ requires access to the $Price$ table in the DB component. Out of all the requests identified, a call to the DB component is initiated from the webserver component.

D. Result Analysis

The experimentation is conducted through saving the failure cases in a case table as shown in Table I. The first row identify different component failures that are monitored and the first column highlights the user activity in the e-commerce web pages [20]. The failure cases are fed into the monitor vm that matches the component vm's for the availability if defined failures.

TABLE III. TIME-CONSUMPTION AND VM-ALLOCATION OF REDUNDANT COMPONENTS

Failed Case	Time (sec)	VM-allocation	State-status
Iteration-1	0.44	0	Not-Preserved
Iteration-2	0.60	6	Preserved
Iteration-3	0.68	7	Preserved
Iteration-4	0.34	4	Preserved

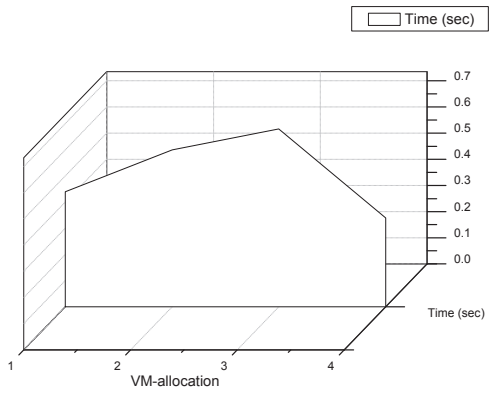

Fig. 3. Time-consumption representation for self-healing in the tested iterations

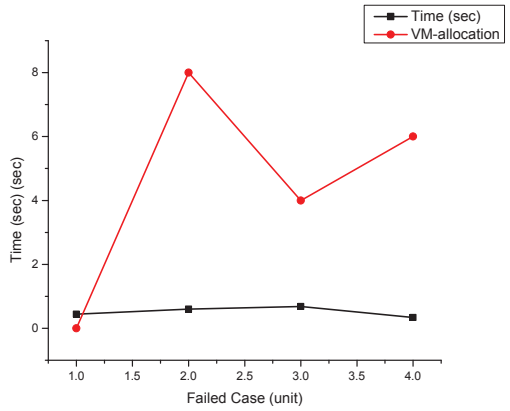

Fig. 4. Vm-allocation and Time-requirement comparison

The experiments contained four iterations where the user conducted a number of pre-defined activities and Monitor was configured in each of the components to detect failures. In 4 iterations of the experiment the failure conditions were triggered with respect to the four criteria described in previous section. Table II highlights the output of the failures in 4 iterations during experiment. The 1 shows an occurrence of a failure in the pre-specified attribute and 0 shows that failure did not occur.

After detection of the failures in Table II, distance between the obtained iteration and pre-defined failed cases are calculated. Exclusive OR is used to identify the distances between each saved failed case and failures of iteration. Among those

the minimum distance is calculated to identify the closed failed case. When the closest case is determined, the solution of that case is implemented for the target iteration and the time consumed for system restoration and self-recovery are recorded. Also time required for vm-allocations are recorded to identify the number of active cloud vm's from the resource pool.

The time consumed for vm allocation and self-healing has been shown in Table III. For every iteration, the time was less than the predefined standard of 0.7 seconds [16]. The table shows that no vm is allocated for Iteration 1 and that no state was needed to be preserved since it was closest to the case of homepage loading for the e-commerce sight, thereby does not require critical saving of states. The performance of the system in terms of high speed self recovery has been highlighted in Figure 3. The time-consumption is respond to vm-allocation has been represented graphically in Figure 4. The results show the desirable low time overhead of the proposed self-healing framework and state preservation of failed software components.

V. Conclusion

This paper has proposed a framework for self-healing mechanism in complex distributed applications. Through the identification of the failures in the processes and components, the proposed framework regenerated the paralyzed components to operate seamlessly. The framework aimed to reinvoke the paralyzed or faulty components by the $Reviver$ module, which initiated the components with the previous state and necessary information provided by the $Monitor$ system. The proposed architecture compared detected faults with pre-defined failure case table through exclusive OR that enabled obtainment of distance between those in each case. The solution of the predefined entry in the failure case table that had the minimum distance with the detected failure was applied since it is the closest to recover from the obtained failed condition.

Empirical evaluation of the experimental iterations show that time required for self healing of the components is below 0.7 seconds in all the four test cases. At the same time minimum distance was calculated as 0-2 units in all the iterations, showing similarity of the cases to real life scenarios. Since cloud is used to aid in rapid allocation of vm-instances, redundancy can be achieved. As a result the distributed software applications can be made more fault tolerant through amalgamation of the proposed self-healing mechanism.

As stated earlier, the proposed framework ensures effective self-healing of distributed components using redundant vm-instances from the cloud. Association of learning methods for the framework to sprawl instances in accordance to component requirements in real time is an area of future research interest.

References

[1] V. Nallur and R. Bahsoon, "A decentralized self-adaptation mechanism for service-based applications in the cloud," *Software Engineering, IEEE Transactions on*, vol. 39, no. 5, pp. 591–612, 2013.

[2] A. Imran, A. U. Gias, R. Rahman, A. Seal, T. Rahman, F. Ishraque, and K. Sakib, "Cloud-niagara: A high availability and low overhead fault tolerance middleware for the cloud," in *Computer and Information Technology (ICCIT), 2013 International Conference on*. IEEE, 2013, pp. 231–237.

[3] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, "Assure: automatic software self-healing using rescue points," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 1, pp. 37–48, 2009.

[4] M. M. Fuad, D. Deb, and M. J. Oudshoorn, "Adding self-healing capabilities into legacy object oriented application." in *ICAS*, vol. 6, 2006, pp. 51–51.

[5] S. Montani and C. Anglano, "Achieving self-healing in service delivery software systems by means of case-based reasoning," *Applied Intelligence*, vol. 28, no. 2, pp. 139–152, 2008.

[6] L. Wei, Z. Yian, M. Chunyan, and Z. Longmei, "A model driven approach for self-healing computing system," in *Computational Intelligence and Security (CIS), 2011 Seventh International Conference on*. IEEE, 2011, pp. 185–189.

[7] N. Cardoso and R. Abreu, "Self-healing on the cloud: State-of-the-art and future challenges," in *Quality of Information and Communications Technology (QUATIC), 2012 Eighth International Conference on the*. IEEE, 2012, pp. 279–284.

[8] M. Bisadi and M. Sharifi, "A biologically-inspired preventive mechanism for self-healing of distributed software components," in *Advanced Engineering Computing and Applications in Sciences, 2008. ADVCOMP'08. The Second International Conference on*. IEEE, 2008, pp. 152–157.

[9] A. Zisman, G. Spanoudakis, J. Dooley, and I. Siveroni, "Proactive and reactive runtime service discovery: a framework and its evaluation," *Software Engineering, IEEE Transactions on*, vol. 39, no. 7, pp. 954–974, 2013.

[10] A. Imran, A. U. Gias, and K. Sakib, "An empirical investigation of cost-resource optimization for running real-life applications in open source cloud," in *High Performance Computing and Simulation (HPCS), 2012 International Conference on*. IEEE, 2012, pp. 718–723.

[11] G. Jung, T. Margaria, C. Wagner, and M. Bakera, "Formalizing a methodology for design-and runtime self-healing," in *Engineering of Autonomic and Autonomous Systems (EASe), 2010 Seventh IEEE International Conference and Workshops on*. IEEE, 2010, pp. 106–115.

[12] J. Park, G. Yoo, and E. Lee, "A reconfiguration framework for self-healing software," in *Convergence and Hybrid Information Technology, 2008. ICHIT'08. International Conference on*. IEEE, 2008, pp. 83–91.

[13] S.-W. Cheng and D. Garlan, "Stitch: A language for architecture-based self-adaptation," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2860–2875, 2012.

[14] C. Parra, X. Blanc, A. Cleve, and L. Duchien, "Unifying design and runtime software adaptation using aspect models," *Science of Computer Programming*, vol. 76, no. 12, pp. 1247–1260, 2011.

[15] H. Song, G. Huang, F. Chauvel, Y. Xiong, Z. Hu, Y. Sun, and H. Mei, "Supporting runtime software architecture: A bidirectional-transformation-based approach," *Journal of Systems and Software*, vol. 84, no. 5, pp. 711–723, 2011.

[16] G. Salvaneschi, C. Ghezzi, and M. Pradella, "Context-oriented programming: A software engineering perspective," *Journal of Systems and Software*, vol. 85, no. 8, pp. 1801–1817, 2012.

[17] A. Imran, A. U. Gias, R. Rahman, and K. Sakib, "Provintsec: a provenance cognition blueprint ensuring integrity and security for real life open source cloud," *International Journal of Information Privacy, Security and Integrity*, vol. 1, no. 4, pp. 360–380, 2013.

[18] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel *et al.*, "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 1–32.

[19] N. Huber, A. van Hoorn, A. Koziolek, F. Brosig, and S. Kounev, "Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments," *Service Oriented Computing and Applications*, vol. 8, no. 1, pp. 73–89, 2014.

[20] H. Ehrig, C. Ermel, O. Runge, A. Bucchiarone, and P. Pelliccione, "Formal analysis and verification of self-healing systems," in *Fundamental Approaches to Software Engineering*. Springer, 2010, pp. 139–153.