J2EE Instrumentation for software aging root cause application component determination with AspectJ

Javier Alonso and Jordi Torres
Barcelona Supercomputing Center
Dept. of Computer Architecture
Technical University of Catalonia
Barcelona, Spain
Email: [alonso, torres]@ac.upc.edu

Josep Ll. Berral
Dept. of Software
Dept. of Computer Architecture
Technical University of Catalonia
Barcelona, Spain
Email: berral@ac.upc.edu

Ricard Gavaldà
Dept. of Software
Technical University of Catalonia
Barcelona, Spain
Email: gavalda@lsi.upc.edu

***Abstract*—Unplanned system outages have a negative impact on company revenues and image. While the last decades have seen a lot of efforts from industry and academia to avoid them, they still happen and their impact is increasing. According to many studies, one of the most important causes of these outages is software aging. Software aging phenomena refers to the accumulation of errors, usually provoking resource contention, during long running application executions, like web applications, which normally cause applications/systems hang or crash. Determining the software aging root cause failure, not the resource or resources involved in, is a huge task due to the growing day by day complexity of the systems. In this paper we present a monitoring framework based on Aspect Programming to monitor the resources used by every application component in runtime. Knowing the resources used by every component of the application we can determine which component or components are related to the software aging. Furthermore, we present a case study where we evaluate our approach to determine in a web application scenario which component/s are involved in the software aging with promising results.**

I. Introduction

Enterprise environments are rapidly changing, as new needs appear. In particular, availability of the information all the time and from everywhere is today a common requirement. To achieve these new challenges demanded by the industry and society, new IT infrastructures have had to be created. Applications have to interact among themselves and with the environment to achieve these new goals, resulting in complex IT infrastructures that need brilliant IT professionals with hard-to-obtain skills to manage them. However, the complexity is achieving such a level that even the best administrators can hardly cope with it.

A recent study[1]showed the average downtime or service degradation cost per hour for a typical enterprise is around US$125,000. Moreover, outages have a negative impact on the company image that could affect profits indirectly. Furthermore, it is now well known that currently, computer system outages are more often due to software faults, and not hardware [1], [2]. Several studies [3], [4], [5] have showed that software aging phenomena is one of the sources of the unavailability. Software aging phenomena refers to the accumulation of errors, usually provoking resource contention, during long running application executions, like web applications, which normally cause applications/systems hang or crash [6]. Gradual performance degradation could also accompany software aging phenomena. The software aging phenomena are often related to others, such us memory bloating/leaks, unterminated threads, data corruption, unreleased file-locks and overruns.

For this reason, the applications have to deal with the software aging problem in production stage, making software rejuvenation techniques [7] necessary. There are two basic rejuvenation strategies: Time-based and proactive-based strategies. In Time-based strategies, rejuvenation is applied regularly and at determined time intervals. In fact, time-based strategies are widely used in real environments, such web servers [8], [9].

On the other hand the proactive strategies system metrics are continuously monitored and the rejuvenation action is triggered when a crash or hang up of the system due to the software aging is an evident provability. This approach is a better technique, because if we can predict the crash and apply rejuvenation actions only in these cases, we reduce the number of rejuvenation actions with respect to the time-based approach.

The effectiveness of these proactive strategies is based on the accuracy of the monitoring system used to collect the system metrics. However, the monitoring systems mainly collect system metrics understanding the applications as *black boxes*, becoming impossible to know which is the root cause of the software aging. They are based on knowing the resource or resources involved in the software aging. For this reason, the main rejuvenation strategy is applying a reboot or application restart. This approach has an important impact over the application availability. New techniques have been proposed to reduce the Mean Time to Recover (MTTR) like Micro-rebooting [10]. Micro-rebooting reduces dramatically the MTTR because they only reboot the faulty component. For this reason, determining the root cause component becomes critical to apply these surgical techniques. However, if we want to determine which component/s are involved in the software aging we need a monitoring framework which allow us to know how many resources are used by every component.

[1]IDC #31513, July 2004

In this paper we present a monitoring framework based on Aspect Oriented Programming (AOP) [11] to monitor the resources used by every application component. Our approach is focused, but not limited, on J2EE architectures, but the same idea could be moved to other languages like C++ [12]. We have focused on J2EE infrastructures because they are the most currently extended for web applications.

Our approach is based on the idea to offer a monitoring solution without need to modify the application or web application server (WAS) source code. In fact, thanks to AOP it is possible to inject our solution to J2EE architectures in runtime. Furthermore, our solution adds a very limited overhead of the original application, allowing us to know, with great detail, the resources used by every component. The idea is to use this framework to establish which component or set of components are consuming more resources to build a resource-component consumption map to help developers and administrators to determine the software aging root cause failure. Moreover, our proposal could be used in development and testing application lifecycle phases to detect misbehaviors, anomalies or to help to optimize the resource usage by the application. Finally, we present a case study where we show how our framework works to decide on where is a memory leak injected in a J2EE web application like TPC-W [13].

The rest of the paper is organized as follows: Section 2 presents the related work. Section 3 describes our framework and the technologies used to develop it; Section 4 presents the experimental case study; and, finally, Section 5 concludes the paper.

II. Related Work

Traditionally, the monitoring tools have been focused on collecting a set of external data from the system like performance, memory consumption, response time, threads used, etc. All of them, understanding the applications or WAS as *black boxes*. As an example, we find several commercial and free solutions like Ganglia [14] or Nagios [15]. Both these systems allow detecting failures in our systems when the failure happens. The detection is based on rules defined by the human system administrators following their experience. However, the effectiveness of these solutions is limited. These tools detect failures but they cannot determine where the error is. The administrator could find the resource or resources involved with the software aging phenomena thanks to these tools, but s/he cannot apply or fix the problem because s/he cannot know where the error is.

In the last years, the new developed applications have introduced tracing code with the objective to help the System Administrator to determine where the error is when the failure happens. However, this approach only offers a post-mortem analysis of the root-cause failure. Moreover, these solutions are not portable to other applications because they are developed ad-hoc and they require a re-engineering work in order to adapt applications to obtain tracing features.

Concerning root cause determination, several approaches have also been reported. The Pinpoint project [16] collects end-to-end traces through the application server with the main goal to determine the more likely component cause of the failures in the system. For this purpose, they use statistical models. They find the components more related with faulty transactions. The idea is collecting all components used by every request and the result of the request (failure or success). This information is used to build a matrix to determine the guilty component or components. Their approach is quite similar to own approach, however they cannot deal with software aging phenomena because they cannot know the resources used by every component. Moreover, the Pinpoint solution has other important limitation: the coupled components. If two components are used always together (very common in J2EE applications) in failure transactions, the Pinpoint framework determines both components with the same probability to be the root cause failure. However, our approach is ready to deal with this situation, understanding every component as independent one.

On the other hand, The Magpie system [17] collects resource consumption by each component to model with high accuracy the system behavior, even of distributed ones. The Magpie approach is the most similar to ours to determine the root cause failure; however, the difference is that we are working at application level and Magpie works at operating system level. Also, the Magpie needs to modify the operating system architecture and our solution is completely independent of the source code increasing the flexibility and adaptability of our solution.

The use of Aspect Oriented programming to monitor the applications is hardly an explored solution. Currently, the most mature solution in this area could be Glassbox [18] which offers a fine-grain monitoring tool. This approach is focused mainly on execution time of every component allowing to detect a big set of failures, however, it is not creating a relationship between the application components and the resources available or used. It is needed to determine the root cause failure of software aging phenomena. Other interesting approach is TOSKANA [19]. it provides an AOP solution for kernel functions but again it is focused on other metrics more than aging-metrics related.

For all of this, it is needed to include monitoring systems which can collect external and internal data from complex systems at runtime to determine the resources used by every component as well as to be able to integrate itself in the system without re-engineering the application or obtain the source code. It is also necessary that these monitoring systems are adaptable and flexible to allow activation or monitoring level change (from application overview to component or even method level or vice versa) at runtime.

III. Monitoring Framework

Before presenting the monitoring architecture proposed we need to present the technologies used to build our approach: Aspect Oriented Programming (AOP) and Java Management Extensions (JMX)[20]. Although it is out of scope of this paper to present in detail both technologies, it is necessary to introduce a brief description of them to make clearer the solution presented in this paper.

A. Used Technology

1) Aspect Oriented Programming: The AOP paradigm allows to isolate the main business logic of the application from secondary functions like logs or authentication. This paradigm increases the modularity, allowing to separate concerns, specifically cross-cutting concerns. *Aspects* is the name of the main concept of the AOP technology. The aspects are composed by two elements: *Advices* and *Join Points*. The advices are the code that is executed when the aspect is invoked: The advice has access to the class, methods and/or fields of the module which the advice invokes. The Join Point is the definition to indicate when the advice will be invoked. We can see the Join Point like a trigger: when the condition is true the Advice is invoked. For this technology's implementation, we have chosen AspectJ [21] because it is a well-known widely used and mature technology. In addition, this technology offers a simple and powerful definition of Aspects like Java class, so the learning curve is quite quick for experienced Java developers. The AOP paradigm is not limited to Java Applications, we can find AOP solutions for C# or C++ like AspectC# [23] and AspectC++ [22] respectively. Furthermore, AOP offers other important and interesting capability for our purpose. AOP allows us to inject code in compile, load or runtime. So, AOP allows us to inject our monitoring framework in runtime without to have access to the source code, allowing even to inject our code over third-part J2EE applications or even legacy Java Applications.

2) Java Management Extensions: The JMX technology offers a set of capabilities to manage and monitor any system component: from devices to Java objects. The JMX is based on 3-level architecture: Probe level, Agent level and Remote Management Level. The Probe level is composed by the probes (called MBeans). Every MBean represents a Java object. The Agent level or MBeanServer is the core of the JMX technology and acts as intermediary between MBeans and the external applications. Finally, the Remote Management Level allows external applications communicate with the MBeanServer via JMX connectors or protocol adapters. So, JMX allows to connect and communicate with Java objects (MBeans) in runtime without modify the application sourcecode allowing to interact with them, transparently.

B. Architecture approach

After present the technologies used to develop our approach, it is moment to present the architecture of our solution. We can divide our approach in four main components: The Aspect Component (AC), the JMX monitoring Agents, the JMX Manager Agent and the External Front-end. Figure 1 shows the components that compose our solution.

1) Aspect Component: Aspect Component is composed by the two elements: The Aspect Component (AC) and the Aspect Component Proxy (AC Proxy). Every application component has an AC associated (thanks to the Join Point definition). The AC has two advices: before and after the application component execution. The idea is to measure every resource before and after a component is used. In this way, we can know how much resource has been used by the component. If the component has a resource consumption bug, the resource available after the execution will be lower than before. To achieve this, the AC communicates (using MBeanServer) to the JMX Monitoring Agents to know the resource status when is demanded. Currently, our architecture is based on a limited set of Monitoring Agents by every resource under monitoring. We have decoupled the JMX Monitoring Agents to the AC (thanks to JMX technology) to increase the adaptability and flexibility of the solution. Currently, if a Monitoring Agent is modified or changed, we don't need to change the AC at all. The MBeanServer capabilities allow the AC to discover new or updated JMX Monitoring Agents. The AC Proxy has the task to create a communication channel between the AC and the JMX Manager Agent. This channel allows the JMX Manager Agent interacts with the AC. From asking some information like how many requests have used by the component to activating or deactivating the AC in runtime. The JMX technology offers a great flexibility and adaptability because we can change the ACs or add new ones and the JMX Manager can discover them by itself and vice versa.

2) JMX Monitoring Agents: JMX Monitoring Agents have the responsibility to access to the operating System and collect resource metrics, AC on demand. So, JMX monitoring Agents usually will be executed before and after every access to every application component under monitoring. Currently, we have developed a limited number of monitors only to show the effectiveness of the approach.

3) JMX Manager Agent: JMX Manager Agent is the core of our proposal. The JMX Manager Agent has the responsibility to collect the metrics by component and build the resource-component map. Furthermore, it has the responsibility to activate or deactivate ACs on demand. For example, to reduce the overhead of the solution or to focus the monitoring over a set of determined objects. The JMX Manager Agent builds the resource-component map and offers a first analysis to establish the most possible root cause component of the software aging in advance. If a software aging has been detected while monitoring the system metrics using a traditional monitoring tool, we can use the JMX Manager Agent (and the rest of our framework) to determine (or at least help to) the component or components involved in the software aging.

4) External Front-end: The *External Front-end* is a simple front-end to allow administrators to communicate with the JMX Manager Agent to know the status of the components in real time or activate new ACs or new JMX Monitor Agents and obtain more details of the application behavior.

C. Root Cause determination Strategy used by JMX Manager Agent

Our current root cause determination mechanism is very simplistic and has to be refined in the future. The main idea is that the component is more aging-related when the

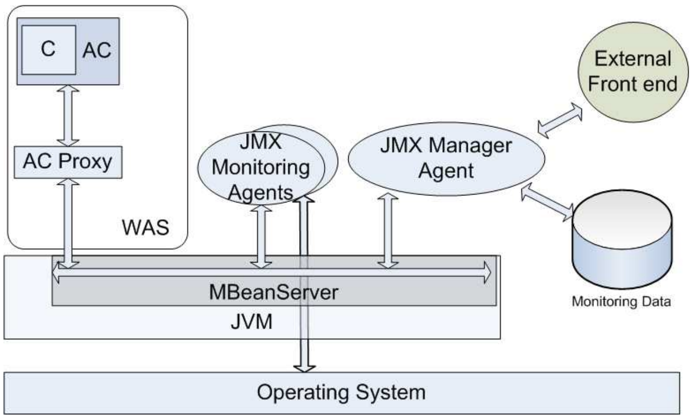

Fig. 1. Components of Monitoring Framework

component resource consumption and the usage frequency is high. In figure 2 we present the core of our current map theory. If a component is very used and the resource usage is high (accumulated along the time) the component increases its probability to become the main aging-component of the application. For example, if we have four components in our application: A, B, C and D. A and B have a memory leak of 100KB in each execution and C and D have a memory leak of 10KB. A and B will be in the right zone of the vertical-axis and C and D in left zone. If A is more used than B then A is in the bottom of the right size (the most suspicious zone) and B in the top. In the same way we can locate the C and D components. Using this analytic approach of the components behavior, the JMX Manager Agent builds the map of root cause aging failure. We have used this approach because the software aging is due an accumulation of aging-errors that usually are consuming resources along the time until their exhaustion. For this reason, we want to know which component is consuming more resources, so which component is more correlated with the aging phenomena.

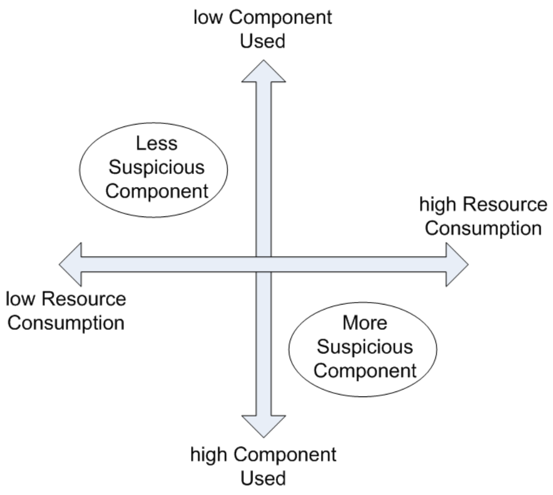
low Component Used
Less Suspicious Component
high Resource Consumption
low Resource Consumption
More Suspicious Component
high Component Used

Fig. 2. Resource Consumption vsComponent Usage map

IV. Experimental Case Study

After present our proposal, we have conducted a set of experiments to evaluate the effectiveness of our approach. Our idea is to test our prototype to determine the cause of a led memory leak. In next subsection we present the experimental environment used in our experiments.

A. Experimental Setup

In this section we describe the experimental setup used in all experiments presented below. The experimental environment simulates a real web environment, composed by the web application server, the database server and the clients machine.

In our experiments, we have used a multi-tier e-commerce site that simulates an on-line book store, following the standard configuration of TPC-W benchmark [13]. We have used the Java version developed using servlets and using as a Mysql [24] as database server. As application server, we have used Apache Tomcat [25]. TPC-W allows us to run different experiments using different parameters and under a controlled environment. These capabilities allow us to conduct the evaluation of our approach to predict the time until failure. Details of machine characteristics are given in Table I.

TPC-W clients, called Emulated Browsers (EBs), access the web site (simulating an on-line book store) in sessions. A session is a sequence of logically connected (from the EB point of view) requests. Between two consecutive requests from the same EB, TPC-W computes a thinking time, representing the time between the user receiving a web page s/he requested and deciding the next request. In all of our experiments we have used the default configuration of TPC-W. Moreover, following the TPC-W specification, the number of concurrent EBs is kept constant during the experiment.

To simulate the aging-related errors consuming resources until their exhaustion, we have modified the TPC-W implementation. In our experiments we have played with Memory resource. To simulate a random memory consumption we have modified a servlet which computes a random number between 0 and N. This number determines how many requests use the servlet before the next memory consumption is injected.

TABLE I
MACHINE DESCRIPTION

	Clients	Application Servers	Database server
Hardware	2-way Intel XEON 2.4 GHz with 2 GB RAM	4-way Intel XEON 1.4 GHz with 2 GB RAM	2-way Intel XEON 2.4 GHz with 2 GB RAM
Operating System	Linux 2.6.8-3-686	Linux 2.6.15	Linux 2.6.8-2-686
JVM	-	jdk1.5 with 1GB heap	-
Software	TPC-W Clients	Tomcat 5.5.26	MySql 5.0.67

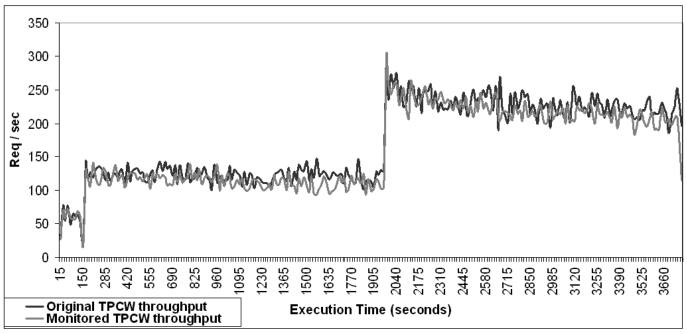

Fig. 3. Throughput of TPC-W under a dynamic workload

Therefore, the variation of memory consumption depends of the number of clients and the frequency of servlet visits. According to the TPC-W specification, this frequency depends on the workload chosen. This makes that with high workload our servlet injects quickly memory leaks, however with low workload, the consumption is lower too. But, again, the average consumption rate would depend on the average of this random variable, with fluctuations that become less relevant when averaged over time. Therefore, we could thus simulate this effect by varying N, and we have decided to stick to only one relevant parameter, N. This error helps us to validate our framework under different scenarios. TPC-W has three types of workload (Browsing, Shopping and Ordering). In our case, we have conducted all of our experiments using shopping distribution.

B. Experimental Results

We have conducted a set of experiments to determine the effectiveness of our approach to monitor the resources consumed by every application component under the experimental environment described before.

1) Framework Overhead: As we presented before, we have injected a set of components (Aspects) to the application code increasing the instructions executed by the computer. So, our monitoring framework has an impact over the performance of the applications. Our first experiment was to evaluate the performance penalty introduced by our framework in an one hour execution of TPC-W with two workload changes. The first two minutes (the warm-up) the workload was 50 Emulated Browsers. During the next 30 minutes the workload was increased to 100 EBs and finally, the last 30 minutes, the workload was 200 EBs. In figure 3 we can observe the throughput obtained by the original TPC-W and TPC-W under monitoring with our infrastructure.

The penalty overhead is quite promising: only 5% of overhead, monitoring all TPC-W application components. In this experiment we didn't inject any memory leak. The response time penalty is quite complicated to evaluate because TPC-W uses a thinking time to simulate the time used by users to read the webpage. This time is random following statistical distribution. For this reason, two executions have different response times. However, the workload (requests by time unit) is constant a long the time.

2) Effectiveness to determine a memory leaking component: After evaluating the overhead introduced by our framework we decided to test our approach under a software aging due to a memory leak in one component of TPC-W. The memory leak was injected as it was described before and we introduced 100Kb of memory leak. We have developed a JMX Monitoring Agent which allows us to know the real size of a Java Object. The real size of a Java Object includes the size of the objects referenced by the object under monitoring. But not the following referenced objects, avoiding a recursively process. In J2EE applications, all objects inherit from superclass and if we apply recursively the process to calculate the size of the object, we find that one object has a indirect relationship with practically all objects of the application. Thanks to this monitoring agent we can know the memory object size at every moment.

We conducted a one hour execution injecting 100KB with $N = 100$ in component A (see figure 4) and the rest of components are not modified. We can observe clearly how the component A is growing in memory size due to the memory leak, becoming clear which is the guilty component of the software aging. While the rest of component sizes are constant a long the experiment consuming a few Kbs, the Object A size is growing from few Kb to MBs consumed during the experiment. In this point our simple mechanism is quite clear, only one component has more memory than the rest of them, concluding that A has the 100% of the responsibility of the software aging.

3) Effectiveness to determine a set of memory leaking components: The next experiment presents the effectiveness of our approach to determine how four components are guilty of the software aging, but different level of responsibility. We have conducted a new one hour execution, however this time, four objects (A, B, C and D) have been modified to inject 100KB following the formula described in the experimental setup section. In figure 5, we present only the four guilty components to reduce the image. We can observe how the four object sizes are increasing along the experiment but at different rate due to the injection mechanism in deed (all objects follow the same injection rate configuration $N = 100$) and the frequency they are used by the clients (EBs). We can observe how the components A and B have the same memory

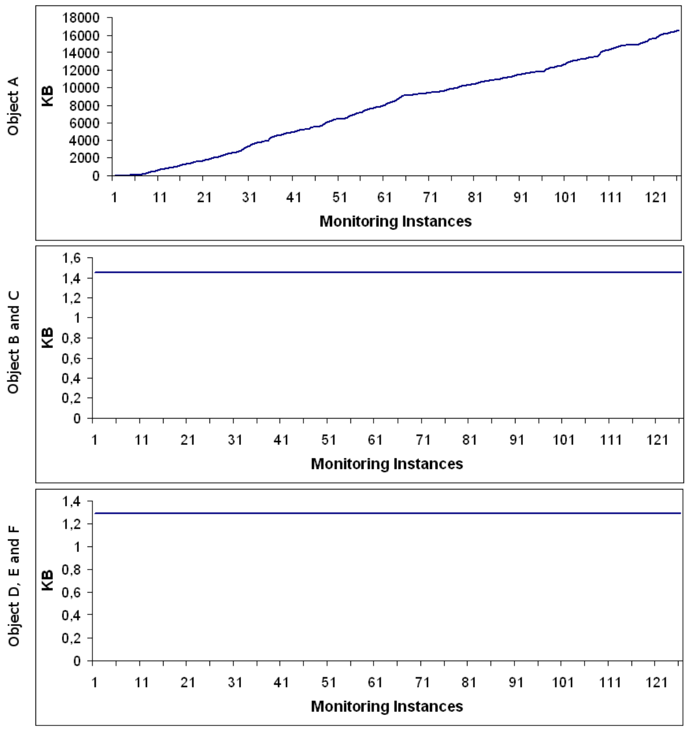

Fig. 4. Injection in component A (100KB)

size after the same period. This fact indicates that both objects have more or less the same frequency usage by the users and, for this reason, in average the same memory leak. They will be in the bottom right zone of the figure 2. However, we can observe how object C has a less memory used, so, object C will be in the top right zone. Finally, we can observe how object D never injected a memory leak because the frequency usage by the users is too low to provoke the injection. For this reason the object D memory usage its maintained constant. The reason is because this object is used not so much in TPC-W shopping workload distribution. It will be in the top right zone. Following our approach objects A and B will be the most suspicious components, after that, object C and finally, object D. Figure 6 shows the composited map by JMX Manager Agent.

4) Effectiveness to determine the root cause failure under different injection sizes: Our approach allows to know how the components are consuming the memory along the execution. After that, we decided to repeat the last experiment, but in the new experiment we injected only 10KB in Object B, 1MB in Object C and D while object A becomes the same with 100KB. The idea is showing how to the memory leak and the usage of the component has an impact over the suspicious level of the component or components to be the root cause of software aging.

In figure 7 we show the memory size of the four objects. We can see again how object D results in constant object size (2KB aprox.) because is used too low. On the other hand, in the last experiment Object C was in third position following

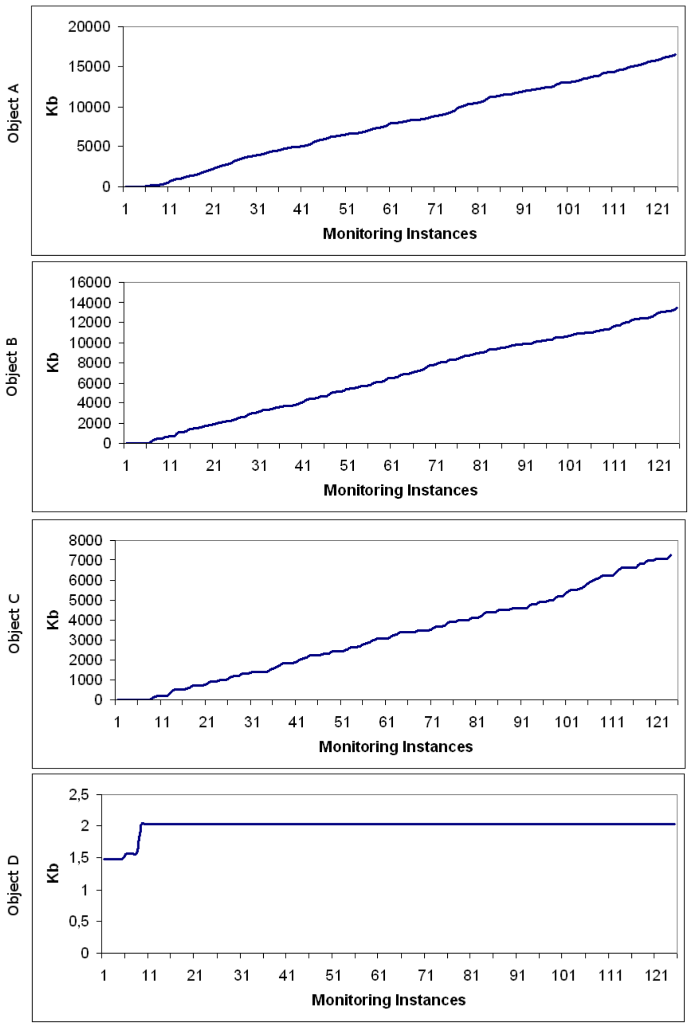

Fig. 5. Detail of injection in 4 components

our root cause determination approach. However, in the new experiment, increasing the size of the memory leak (from 100Kb to 1MB) has becoming on the most important reason of the software aging. Objec A continues being an important factor of the software aging (second position) and Object B, has also important impact over the memory consumption but now with a lower memory leak (from 100KB to 10KB), it is in third position.

V. Conclusion

In this paper we have presented a monitoring framework for detecting software aging root causes, using the Aspect Oriented Programming and Java Management Extensions technologies. Our methodology allows monitoring the applications without altering their source code and injecting the observers in runtime, being able to connect or disconnect them on demand.

For this case of study, we focus on a specific kind of software aging: memory leakage. We present a theoretic map where, depending on the components observed behavior, we

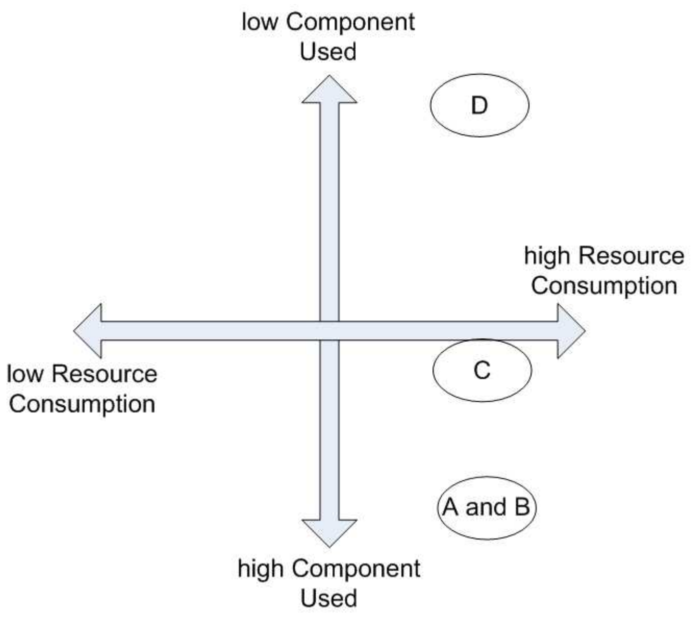

Fig. 6. Resource Consumption vs Component Usage map composed by JMX Manager Agent

can determine the component that with high probability is the root cause of the resource consumption.

Our experimentations show that our method allows determining, given an aging-error, the most suspicious components, helping designers and system operators to look for the real cause and then fix the problem.

In our future work, we focus on the application of this framework and methodology towards other software aging causes, like CPU and thread leaks among others, and also improve the determination method looking for more *intelligent* decision makers in front of different software aging symptoms.

ACKNOWLEDGMENT

This research work has been supported by the Spanish Ministry of Education and Science (projects TIN2007-60625) and by the Generalitat de Catalunya (2009-SGR-980). R. Gavaldà is partially supported by the EU PASCAL2 Network of Excellence and by the MICINN SESAAME project, TIN2008-06582-C03-01/TIN.

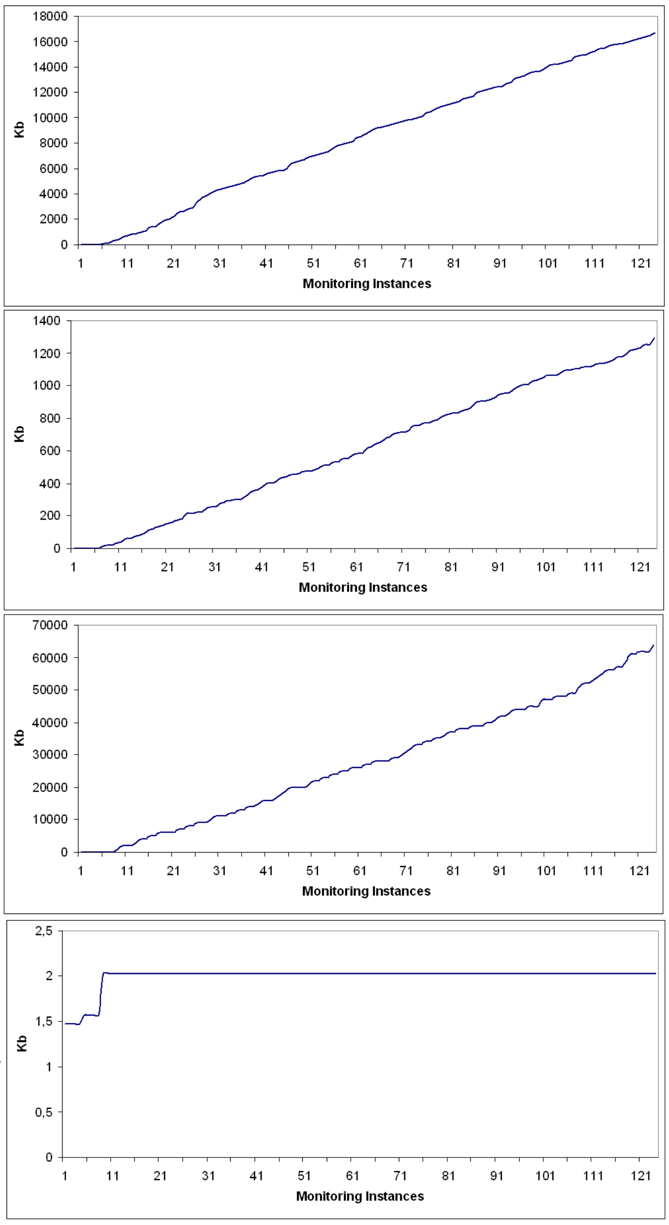

Fig. 7. Determination of Root failure in 4 different injections

REFERENCES

[1] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. *Why do internet services fail, and what can be done about it?*.In 4th USENIX Symposium on Internet Technologies and Systems (USITS'03), 2003.

[2] S. Peret and P. Narasimham. *Causes of Failure in Web applications*. Technical Report CMU-PDL-05-109, Carnegie Mellon Univ, Dec. 2005.

[3] K. Vaidyanathan, R. E. Harper, S. W. Hunter, and K. S. Trivedi. *Analysis and Implementation of Software Rejuvenation in Cluster Systems*. ACM Sigmetrics 2001/Performance 2001.

[4] K. S. Trivedi, k. Vaidyanathan and K. Goseva-Popstojanova. *Modeling and Analysis of Software aging and Rejuvenation*. IEEE Annual Simulation Symposium, April 2000.

[5] T. Dohi, K. Goseva-Popstojanova and K. S. Trivedi. *Analysis of Software Cost Models with Rejuvenation*. IEEE Intl. Symposium on High Assurance Systems Engineering (HASE 2000).

[6] M. Grottke, R. Matias Jr. and K.S. Trivedi *The Fundamentals of Software aging* In Proc. 1st Int. Workshop on Software Aging and Rejuvenation. 19th Int. Symp. on Software Reliability Engineering, 2008.

[7] Y.Huang, C.Kintala, N.Kolettis and N. Fulton. *Software Rejuvenation: Analysis, Module and Applications*.Proceedings of Fault-Tolerant Computing Symposium, FTCS-25, June 1995

[8] K.Vaidyanathan and K.Trivedi *A Comprehensive Model for Software Rejuvenation*. IEEE Trans. On Dependable and Secure Computing, Vol, 2, No 2, April- 2005

[9] S. Garg, A. van Moorsel, K. Vaidyanathan and K. Trivedi *A Methodology for Detection and Estimation of Software Aging*. Proc. 9th Int'l Symp. Software Reliability Eng., 1998.

[10] G.Candea, E.Kiciman, S.Zhang and A.Fox *JAGR: An Autonomous Self-Recovering Application Server*. Proc. 5th Int Workshop on Active Middleware Services, Seattle, June 2003

[11] G. Kiczales et al. *Aspect Oriented Programming*. Lecture Notes in Computer Science, Vol. 1241, pp. 220-242, Springer 1997.

[12] D. Mahrenholz, O. Spinczyk, and W. Schröder-Preikschat *Program*

Instrumentation for Debugging and Monitoring with AspectC++. Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2002.

[13] TPC-W Java Version *http://www.ece.wisc.edu/ pharm/*.

[14] [Web-site at Dec. 2009] http://ganglia.info

[15] [Web-site at Dec. 2009] http://www.nagios.org

[16] M. Chen, A. Accardi, E. kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. *Path-based failure and evolution management*. Proc. of the 1st Symp. NSDI'2004.

[17] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier *Using Magpie for request extraction and workload modelling*. Proc. of the 6th Symp. OSDI'2004.

[18] [Web-site at Dec. 2009] http://www.glassbox.com

[19] M. Engel and B. Freisleben *Supporting autonomic computing functionality via dynamic operating system kernel aspects*. Proc. of the 4th Intl. Conf. on Aspect-Oriented software development, pp. 51-62, March 14-18, 2005.

[20] [Web-site at Dec. 2009] http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/

[21] G. Kiczales et al. *An Overview of AspectJ*. Proc. European Conf. for Object-Oriented Programming (ECOOP 2001). Lecture notes in Computer Science, Vo. 2072, pp. 626-657, 2001.

[22] [Web-site at Dec. 2009] http://www.aspectc.org/

[23] H. Kim *AspectC#: An AOSD implementation for C#*. Master Thesis Dissertation, Trinity College Dublin, 2002.

[24] MySQL Data Base server *http://www.mysql.com/*.

[25] Apache Tomcat Server *http://tomcat.apache.org/*