Abstract—The growing complexity of software systems is resulting in an increasing number of software faults. According to the literature, software faults are becoming one of the main sources of unplanned system outages, and have an important impact over company benefits and image.

For this reason, a lot of techniques (such as clustering, fail-over techniques, or server redundancy) have been proposed to avoiding software failures, however they still happen. They can be divided in three categories: permanent, intermittent and transient failures. The first type are easy to fix, but the other two categories are difficult to reproduce and fix, and become a real problem for developers and system administrators.

Here we propose a new framework for predicting in real time the time until crash of a clustered application, using machine learning techniques. Our framework allows recovery of the potentially crashing server using a clean automatic recovery and avoiding losses of new or on-going user requests. Furthermore, our framework offers a valuable feedback to the developers by assigning suspicion levels (of provoking the crash) to every component of the application. We have tested our framework in a three-tier web J2EE clustered application achieving promising results.

I. INTRODUCTION

A. Problem Statement and Importance

As the complexity of software systems continues to grow, so increases the difficulty of managing them. Moreover, our current and growing reliance on these software systems to manage critical and ordinary tasks in our life requires these software systems not only offer an acceptable performance but also continuous availability. To meet these society needs, more skilled developers and administrators are needed to maintain these complex and heterogeneous software systems, resulting in a large fraction of the total cost of ownership (TCO) of these systems. Furthermore, the complexity of these software systems is reaching the maximum complexity that human experts can manage [1], so it becomes necessary to develop new systems which can self-manage with minimum human intervention, as described in [2].

The Autonomic Computing initiative [3] groups a lot of research efforts in different areas to achieve these goals. Research efforts are divided in four directions: self-configuring, self-optimizing, self-protecting, and self-healing. Self-configuring addresses the dynamic changes of the environment where the systems are deployed, allowing the system to adapt itself to these environmental changes. Self-optimizing focuses on maximizing resource usage and performance. Self-protecting systems can defend themselves from external or internal malicious attacks. Finally, self-healing systems, the focus of this paper, are able to predict, detect, diagnose, and repair anomalies due to faults, and avoid failures.

Every year, more industry and academia efforts are focused on maintain the systems running without downtime as much time as possible, trying different approaches (such as clustering, fail-over systems or redundancy servers) in search of the famous three nines of availability. However, systems still suffer unplanned downtime periods. Usually, the most expensive (not only in money) downtime is the unplanned kind, because they impacting negatively on company benefits, image, and prestige. This impact is even more important for companies and services that have only on-line presence, because while the service is down, the company directly does not exist and the potential customers move on to competitors.

There are not that many studies on the reasons of unplanned system outages. According to the literature, we can divide the outages causes in three groups: hardware, human, and software. In mid-80s and early 90s, the main causes of outages were the hardware and human errors, around 40% each, and software errors represented around 20% of failures [6], [7]. However the figures have changed from the 90s to now. In more recent studies [8], [9], [10], [11], hardware errors are decreasing from 40% until around 20%, while software errors
are significantly increasing during the same period from 20% until 40%. This evolution shows clearly the relationship of the complexity of the software systems and their vulnerability to bugs and other software failures [12], [13], [14]. On the other hand, the percentage of human errors remained more or less constant during the time. The main reason seems to be the creation and effectiveness of new administration and monitoring tools that, at least until now, have helped administrators to cope with the growing system complexity. Since human errors do not seem to decrease either there are important challenges to address in this area, but they are out of scope of this work.

Software bugs are faults and they are unavoidable in any software system. We can divide the software bugs in two main categories: Bohrbugs and Heisenbugs [6]. Bohrbugs are deterministic and (relatively) easy to fix using traditional testing or debugging techniques by developers. Heisenbugs, on the other hand, manifest themselves and disappear nondeterministically, so they are much more difficult to track down, or even to reproduce. To be precise about causes, effects, and observable effects, we use the terms understand fault, error/bug, and failure following the definitions in [15]. Briefly, a fault is the cause of an error; it is active when it causes an error, otherwise it is a dormant fault. An error or bug is a part of the total state of the system that may lead to its subsequent service failure. And a failure is a visible erroneous operation of the system, probably due to an error.

Software failures can be divided in three main categories according to their impact over the system behavior in time: permanent, intermittent, and transient failures. According to the literature [6], [7], [9], [10], the most complex failures to overcome are transient and intermittent failures caused by an anomalous resource consumption by the system (memory, CPU, connectivity, etc.) due to Heisenbugs, because their behavior is non-deterministic. These type of failures resulting on undesirable behavior of the system or even in hang ups or crashes. However, resource exhaustion could be caused by different reasons, and not all of them could be defined as anomalies or faults. If the system is overloaded due to an excessive workload, it is possible that one or more resources are overloaded and the system becomes down. In this case, we face simply a system with insufficient resources for its demand. But if resource consumption cannot be explained by the workload or the system behavior, we are clearly dealing with an anomaly or fault. Degradation of the running systems for no obvious (possibly nondeterministic) cause has been called software aging. This phenomena has been found in any type of complex enough systems such as Telecommunication Systems [16], web-servers [17], [18], enterprise clusters [19], OLTP systems [20], spacecraft systems [21], and even in military systems [22] with critical consequences such as loss of lives.

B. Our Contribution

In this paper we present a machine learning based run-time framework to predict the time until a crash due to resource exhaustion in clustered web applications. Our mechanism monitors the state of the system and, based on system activity reports (SARs) such as CPU usage, Memory consumption, number of threads, etc., tries to predict when it will run out of some critical resource. To this end, our approach uses machine learning techniques of two flavors. One, numerical, to predict the time until the system will crash. Another, categorical: our system is able to flag “red-light” alarms when a crash is imminent, possibly triggering automatic recovery procedures or calling for human intervention. We have used machine learning algorithms with low computation needs because our framework has to be able to predict the time until failure numerically and the alarms on-line, in real-time. For prediction, we start only from the system level metrics (SARs) and apply an enrichment process to obtain derived variables such as resource exhaustion speed. We explain this process in detail in section III.

If the framework determines that a crash is near, it triggers a recovery action applies a clean restart of the critical-looking node of the cluster. The clean restart avoids missing any ongoing or new request to the cluster using an “intelligent” load balance based on traditional Linux Virtual Server (LVS) [23]. We allow the problematic server to complete on-going requests, but stop sending any new requests to it. When the server is empty a clean restart is applied. This approach was presented in detail in a previous work [24] using virtual machines; in this paper, however, we have migrated the technique to clustered applications.

Finally, the framework offers a post-mortem analysis of potential root-causes of the failure, assigning a level of suspicion to every application component. We use Aspect-Oriented Programming [25] to monitor each component’s behavior in run-time without having to modify the application source code. This feature should be extremely useful to the developers that have to fix the bug, since fixing an intermittent bug in a complex and large-scale software system usually requires a lot of high-skilled human hours [26].

The rest of the paper is organized as follows: Section 2 presents the related work, Section 3 describes in detail our mathematical model and our prediction strategy; Section 4 describes our framework and its prediction and recovery techniques and outlines their simplicity; Section 5 presents the experimental setup in our experiments. In Section 6 the crash prediction technique is presented and its results of our experimental study of the framework to predict the time until failure and its recovery techniques; Section 7 describes the technique to determine the suspicious component level developed and presents the preliminary experimental study and finally, Section 8 concludes the paper.

II. Related Work

The idea of modeling resource consumption and from here forecast system performance is far from new. A lot of effort along this line has been concerned with capacity planning. In [27], an off-line framework is presented to develop performance analysis and post-mortem analysis of the causes of Service Level Objective (SLO) violations. It proposes the
use of TANs (Tree Augmented Naive Bayesian Networks), a simplified version of Bayesian Networks (BN), to determine which resources are most correlated with performance behavior. In [28], Linear Regression is used to build an analytic model for capacity planning of multi-tier applications. They show how Linear Regression offers successful results for capacity planning and resource provisioning, even under variable workloads. Other works such as [29], [30] present different techniques to predict resource exhaustion due to a workload in a system that suffers software aging. In these two works they present two different approaches: in [29], authors use a semi-Markov reward model using the workload and resource usage data collected from the system to predict resource exhaustion in time. In [30], authors use time-series ARMA models from the system data to estimate the resource exhaustion due to workload received by the system.

Concerning failure and critical event prediction, several studies have been conducted in other areas such as Telecommunications [31], [32], [33]. However, these prediction techniques are not sufficient to manage the complex and varied system health data in computer systems. Some interesting works have addressed the important task of predicting failures and critical events specifically in computer systems. In [34], the authors present a framework to predict critical events in large-scale clusters. They compare different time-series analysis methods and rule-based classification algorithms to evaluate their effectiveness when predicting different types of critical events and system metrics. Their conclusion is that different predictive methods are needed according to the element that we want to predict. Paper [35] presents an online framework that determines whether a system is suffering an anomaly, or a workload change, or a software change. The authors use Linear Regression. The idea is to divide the sequence of recorded data into several segments. A segment is divided in two when no single Linear Regression model gives acceptable error (3%) on the whole segment. If on each of the two resulting segments there is some model with acceptable error, they determine that a software change has occurred at their cutpoint. If for some period it is impossible to obtain any Linear Regressions with acceptable error at all, the conclusion is that the system is suffering some type of anomaly during that period. Their approach is complementary with ours, because the underlying assumption is that, except on transient anomalies and between software changes, the system admits a static model, one that does not degrade or drift. On the other hand, we concentrate on systems that can degrade, i.e., for which a model valid now will not be valid soon, even under the same workload.

Concerning root cause determination, several approaches have also been reported. The Pinpoint project [37] 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. The Magpie system [36] 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 determinate the root cause failure; however, the difference is that we are working at application level and Magpie works at operating system level. Also, machine learning techniques have been used in several studies to determine root causes of failure. In [38], a mechanism based on decision trees is presented. They build decision trees using request traces from periods in which user-visible failures are presented to predict future failures. In [39], authors present an approach based on machine learning techniques to determine the most relevant metrics and localize the anomalies in large-scale clusters. That work focused on High performance computing (HPC). They use principal component analysis (PCA) to determine the most relevant metrics and outlier detection techniques to identify the nodes that have problems.

III. OUR MODELLING ASSUMPTIONS AND PREDICTION STRATEGY

In this section we describe the mathematical model and assumptions on which our prediction strategy is based. We assume a set of predefined measurements $M_1, M_2, \ldots, M_n$ of a system. These measurements are taken at different points in time: $M_{i,t}$ is the value of $M_i$ seen at time $t$. Examples could be available memory, response time, number of active processes, cpu usage, requests per unit time, etc. For simplicity of notation, we will assume that the $t$’s at which we perform the measurements are $t = 0, 1, 2, \ldots$; the framework is easily extended otherwise (e.g., if the times are arbitrary date/time pairs, and not necessarily equally spaced).

Systems are, in theory, deterministic, so the values and evolution of measurements are completely determined by the incoming workload and of the system’s internal state. But, in practice, the relation between workload, internal state, and measurements is so horrendously complex that we can only think of $M_{i,t}$ as random variables, i.e., as values coming from some (unknown) probability distribution, and deal with them statistically.

We denote with $R_{i,t}$ the expected value of $M_{i,t}$. The choice of the name $R_{i,t}$ is not neutral: we think of $M_{i,t}$ as measuring some kind of resource $R_i$ over time. Sometimes, the measurement is totally accurate (e.g., when measuring the amount of available memory, number of alive threads or sockets, etc.), so we have $R_{i,t} = M_{i,t}$. Sometimes, however, the measurement $M_{i,t}$ has noise in itself, or the “resource” that it measures is not as well defined as to be directly observable. Consider, e.g., the “response time for a request”, of which we can only take means over interval of time.

The difficulty, of course, is that abstract resource $R_i$ varies over time; otherwise we could simply approximate it by averaging $M_{i,t}$ over sufficiently long times. Let us call $\delta_{i,t}$ the average variation of resource $R_i$ from time $t-1$ to time $t$, i.e.,

$$R_{i,t} = R_{i,t-1} + \delta_{i,t} \quad (1)$$

Note that it is in $\delta_{i,t}$ were we hide most of the low-level system’s complexity that we are unable to measure, explain,
or analyze. For a workload that can be processed with the available resources, and for a non-faulty system, it is in many cases reasonable to make the following hypothesis:

**System stability hypothesis:** Assuming the workload characteristics remain constant, the system will converge to a set of values of \( R_{i,t} \) depending on the workload only, after sufficiently long time \( t \). In particular, \( \delta_{i,t} = 0 \).

Note that this hypothesis does not imply that the measurements \( M_{i,t} \) remain constant over \( t \). They may suffer some random fluctuations but these, in average, tend to cancel out around \( R_{i,t} \).

That is, for a given workload \( W_i \) we can associate a value to each measurement that represents the value to which it will converge under that workload. As an example, suppose that an application uses a minimum of 200Mb, plus 1Mb for each request it is processing. If the workload consists of 50 simultaneous requests, the application uses 200+1*50=250Mb. If it later increases to 200 simultaneous requests, we would expect memory usage to go up to 200+1*200=400Mb. The hypothesis implies that if the load goes down again to 50 requests, the system gradually returns to using 250Mb, i.e., no memory is lost on the way.

Now, we consider failures occurring for two reasons:

1) The incoming workload \( W \) is too large to be dealt with available resources. Such failures can, in theory, be anticipated by estimating the resources required by the current workload and predicting that they fall outside some “feasible region”.

2) The given workload can, in principle, be handled by the infrastructure, but the system degrades over time. If such degradation can be observed from the measurements, this is to say that \( R_{i,t} \) does not remain constant over time \( t \), that is \( \delta_{i,t} \) is not zero in general. In this paper we propose a strategy for dealing with this kind of failures: By monitoring the speed at which \( R_{i,t} \) varies (equivalently, monitoring the evolution of \( \delta_{i,t} \)), we can estimate a time \( T_{fail} > t \) such that \( R_{i,T_{fail}} \) is likely to fall out of the feasible region, i.e., failure will occur.

We concentrate on type 2) failures, and consider under which condition we can make any prediction at all. For sure, approximate prediction is possible when degradation occurs at a roughly constant speed, that is, \( \delta_{i,t} \) remains constant (but nonzero) over reasonable periods of time, say, to a value \( \delta_i \). For \( T > t \), we then have

\[
R_{i,T} = R_{i,t} + (T - t) \cdot \delta_i.
\]

Suppose additionally that we know a value \( R_{max} \) such that when \( R_i \) reaches \( R_{max} \), failure is likely to occur. We then have \( R_{i,T_{fail}} = R_{max} \) and therefore can estimate \( T_{fail} \) as:

\[
T_{fail} = t + \frac{R_{max} - R_{i,t}}{\delta_i}.
\]

Under this constant-degradation-speed hypothesis, one should therefore make the prediction that failure will occur around time \( T_{fail} \). For this reason, we propose the following: given a reliable estimate of the current degradation speed, we can still use the equation above to predict some failure time, assuming no change in the speed. In other words, our system will predict “if the system keeps degrading as right now, then failure should occur by time \( T_{fail} \);” if the speed seems to increase or decrease later on, it will update its prediction accordingly.

To actually apply equation (3), we need two parameters that affect the prediction linearly: the inverse of the current speed, \( 1/\delta_i \) and the value of \( R_{max} \). \( \delta_i \) can be estimated as the average of most recent observations, with some smoothing (see next paragraph). \( R_{max} \) is, in principle, not directly observable. However, this is where machine learning techniques become useful: it will be estimated indirectly from data from past crashes, e.g., in the form of a regression coefficient of a Linear Regression model.

Finally, in order to estimate measurements and their speed in a noise-tolerant way, we should use one of several techniques to smooth them out over some period of time. A common one is to take averages over a window of pre-specified length from the past. We used instead the EWMA (Exponentially Weighted Moving Average) method described e.g. in [40], which assigns more weight to more recent points.

The discussion above has dealt with a single resource, for simplicity of exposition. When several resources are monitored and can contribute to the crash, the models and regression problems become multivariate but the basic principle remains the same.

**IV. THE PREDICTION AND RECOVERY FRAMEWORK**

In this section we present our framework in detail. Figure 1 presents the basic architecture and components of our approach. The framework has been designed taking into account that we are focusing on self-healing for clustered web applications; we consider that these applications are formed by a load balancer, and a set of web application servers (nodes), where the applications are deployed, and a centralized or distributed data base. It is based on the MAPE Architecture defined by Kephart and Chess in [3], that defines a cyclic system with four stages: Monitoring, Analyzing, Planning, and Executing. Our framework covers all four stages, and we describe each of them next.

![](Fig. 1. Basic description of framework components)

**A. Monitoring Subsystem**

The monitoring subsystem’s task is to obtain the SARs metrics to obtain instantaneous descriptions of the system state. It is installed in every node of the cluster that we want to include in our self-healing, as shown in figure 1.

Currently, our monitoring system is divided in two main agents: The system-level monitoring agent and the application component monitoring agent. The system-level monitoring
agent collects metrics from the whole system and specifically related with the web application server under monitoring such as memory used by the application server, CPU used, number of threads running on the system, throughput, response time of the server, etc. Moreover, our monitoring system is focused on J2EE web application servers. The J2EE application servers run over the Java Virtual Machine (JVM) instead of the Operating System. Due this reason, this agent also monitors the JVM Heap Memory status. The current version of the system-level monitoring agent is based on the well-known and used monitoring tool called Nagios [41]. The Nagios monitoring tool offers a decentralized, scalable and easy to upgrade monitoring framework used in several Computer centers. We have added some monitoring plug-ins to add more metrics to Nagios to obtain more data from the system.

On the other hand, the application component monitoring agent monitors the resources consumed by every application component, by every object or class of the application running over the web application server, and even any object or class of the web application server itself. This monitoring agent has to monitor inside the application; however, to propose an extensive and potentially general solution, our approach had to be little intrusive and in any case we could not modify the source of the application. For this reason, we have decided to use JMX and Aspect Oriented Programming (AOP) technologies allowed in Java to cover our requirements. We have used the object monitoring system presented in [42]. AOP allows to inject code in configurable points of the code in compile, load or running time. We have used this possibility to inject the monitoring agent to capture resources consumed by every component, following the philosophy presented in [36]; we work however at different level, obtaining a fine-grain application monitoring system.

The monitoring system captures the state of the system every $N$ seconds. In our experiments, $N$ was fixed to 15 seconds and the data, in the form of a tuple which represents the system state, is sent to the Analysis subsystem.

B. Analysis Subsystem

The Analysis subsystem is divided in two modules as is showed in figure 1: the enriching system and prediction system. In our current prototype version of the framework, we have a centralized Analysis subsystem, although definitely it should be easy and desirable to implement a decentralized analysis subsystem in the future, to reduce the implied bottleneck. The enriching system receives the monitoring data from the monitoring subsystem as is presented in figure 2. The enriching system generates a subset of derived metrics to add to the tuple of the raw metrics. The derived metrics are needed to build a more detailed situation of the node state. For instance, we calculate how quickly the resources are been consumed (the consumption speed of the resources) or the EWMA of the metrics as was presented in the previous section.

After the enriching process has processed the captured metrics, the new tuple is sent to the prediction system. The prediction system is a trained model based on a two machine learning algorithms which are present in detail in section VI. This trained model has two responsibilities. It has to predict the time until crash or failure due to resource exhaustion and in numerical results (in seconds) and it has also to determine whether any node is far or close to the crash emitting a “red-light” alarm. We have divided the status of the system in three levels: Green, orange and red. The red zone means that the system is 5 minutes away from the crash, the orange zone is that between 5 and 10 minutes from the crash, and the green zone means that no crash is expected in less than 10 minutes. This division has been decided for our experiments; these numbers can of course be parametrized and train a new model using other parameters.

C. Planning Subsystem

When the prediction (the red-orange-green zone and the time until failure) for every node has been calculated, they are sent to the Planning subsystem. Currently, our planning subsystem is quite simple and uses a simple threshold to trigger a recovery execution. However, our plans are to investigate how to introduce some Artificial Intelligence inside the planning subsystem to take the best option every time, avoiding useless recovery actions when they are not needed and scheduling recovery actions when more than one node has to be rebooted by the system, to maximize the utilization the cluster. The current preliminary version uses a simple threshold based on “red light” alarm is triggered by the prediction system and the time until failure is under 10 minutes to crash.

D. Execution Subsystem

The execution subsystem is responsible of applying the recovery action decided by the planning subsystem. Our current execution subsystem is divided in two parts: the recovery manager and the recovery agents. The recovery agent is installed on the load balancer/s and a recovery agent is installed in every node. The execution subsystem has been developed to avoid any on-going or new request and is integrated with the LVS load balancer used. The manager is conscious of which nodes are scheduled for rebooting and whether it has in-process requests or not. When a node scheduled for rebooting has finished its last ongoing request, the reboot is started. Also, the new requests are sent to other nodes. This system was presented in our previous work in [24]; see that reference for more details about the recovery subsystem.

In figure 2, we summarize how the different elements of the framework work together.

V. EXPERIMENTAL SETUP

In this section we describe the experimental setup used in all experiments presented below. Figure 3 shows the experimental environment. We can see how we have a simple cluster of two nodes for our experiments and a load balancer. The analysis subsystem and planning subsystem are in an external centralized machine, however in a real environment the best option would be that the analysis subsystem will be distributed in every node and only the planning subsystem will be centralized.
to take decisions taking into account all the information related to all nodes.

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 [43]. We have used the Java version developed using servlets and using as a MySQL [44] as database server. As application server, we have used Apache Tomcat [45] in cluster. 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. As a load balancer we have used LVS as we presented before. Details of machine characteristics are given in Table I.

| TABLE I | DETAILED MACHINES DESCRIPTION |
|-----------------|------------------|------------------|------------------|
| Hardware        | Clients           | Load balancer    | Application Servers | Database server |
| Operating System| Linux 2.6.8-2-686 | Linux 2.6.8-3-686| Linux 2.6.15      | Linux 2.6.8-2-686 |
| JVM             | jdk1.5 with 1024MB heap | jdk1.5 with 1024MB heap | -                | -                |
| Software        | TPC-W             | LVS              | Tomcat            | MySql            |
| Clients         | 5.5.26            | 5.0.67           |                   |                  |

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 [43]. We have used the Java version developed using servlets and using as a MySQL [44] as database server. As application server, we have used Apache Tomcat [45] in cluster. 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. As a load balancer we have used LVS as we presented before. 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. Because we are working on a clustered application, we have to configure Tomcat in the cluster to guarantee the consistence of the sessions using a session replication technique. This means that every application server shares its sessions with the rest of servers, so that any user can be redirected to different servers during the same session without any user visible disruption in the session. Between two consecutive requests from the same EB, TPC-W computes a thinking time (uses a negative exponential distribution with 7 seconds average), representing the time between the user receiving a web page s/he requested and deciding the next request. This value is generated just before every request and the maximum value can be configured; in 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 a transient failure that consumes resources until their exhaustion, we have modified different classes of the TPC-W implementation depending of the experiment (we indicate the class modified in every experiment). The modified servlet computes a random number between 0 and N. This number determines how many requests use the servlet before the next memory leak is injected. Therefore, the variation of memory consumption depends of the number of clients and the frequency of servlet visits. Of course, in average, there be a memory leak injection every N/2 requests. 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. TPC-W has three types of workload (browsing, Shopping and Ordering). In our case, we have conducted all of our experiments using shopping distribution.

VI. CRASH PREDICTION

In this section we present our crash prediction model and a discussion about our approach and present some experiments which show the effectiveness of our proposal. In our previous work [46], [47] we have conducted several comparisons about different machine learning algorithms to determine the best option for our proposes. In [46] we conducted a case study to evaluate three well-known machine learning algorithms: Linear Regression, RepTree and M5P; the last two are variants of the decision tree models. After the analysis of this data, we could conclude that M5P, a mix of decision trees and Linear regression, was the best option if we measured the accuracy according to the Mean Average Error (MAE). However, after discussions with several system administrators, we concluded that a crash prediction must be more informative than a number: it must provide clear and reliable alerts when the
crash is imminent. For this reason, in [47] we analyzed the same algorithms taking into account this second goal. Additionally, we decided to add more system data to further improve our prediction accuracy. In our experiments we were injecting memory leaks from a J2EE application running inside the JVM, while we were monitoring the Memory Consumption by the application server. This approach was partly inadequate because we were injecting memory leaks in the Heap of the JVM. For this reason we decided to monitor every Heap zone and its evolution in time. After this new analysis, we concluded that M5P continued to be best option to predict the numeric time until crash. In the same work, we introduced the idea of the green-orange-red zone to predict the critical event (the crash of the system). Since the prediction then becomes categorical rather than numerical, we used three common machine learning algorithms with categorical output (J48, Naive Bayes and IBk) and observed that the best results were obtained by J48, an implementation of the C4.5 tree induction algorithm.

A. Experimental Results

In this subsection we present an abstract of the experiments conducted using the M5P and J48 in the on-line and real time framework. Due to space limitation, the reader is invited to check [46], [47] for more details on the algorithms and results.

1) (Numeric) time until crash prediction: Our first approach was to predict the time until crash using the M5P algorithm. To compute the effectiveness of our approach we have conducted two experiments using different environments. In the first experiment, we modified one servlet of the TPC-W application to inject a memory leak of 1MB. The times at which memory is injected are computed by choosing, after each injection, a random number between 0 and N (where N is the injection ratio) and waiting for that many requests before injecting again. To make the system more unpredictable, we conducted another experiment where we changed the memory injection ratio every 20 minutes and we test the model trained using M5P algorithm. In figure 4 we can observe the evolution of the memory consumption speed.

The slope represents different ratios, for four different intervals of 20 minutes each. During the first 20 minutes we do not inject any memory leak; however, we can observe the warm up of the Tomcat. In the second 20-minute interval we use $N = 30$, which means that on average we insert a leak every 15 requests. After that we increase the aggressiveness of the memory leak using $N = 15$, and later reduce the bug drastically by using $N = 75$. This changing environment should force our system to recompute the time until crash as the observed leak rate varies.

In figure 5 we present the time until crash predicted by our model trained using different data sets from the testing set. We have used a constant memory injection ratio for training purposes. We have used different workloads but we present only one workload (100EBs) to simplify the data presentation and reduce number of the graphs.

As figure 5 shows, the M5P method produces promising results. First, we can observe clearly how the predictor at the start of the experiment is confused by the warm up process and the fact that during the first 20 minutes we do not inject any memory leak. For this reason, the predictor determines that the time until crash is very big, essentially infinite. After that, when we start to inject memory leaks at 0-30 requests ratio, the predictor reacts after a while and starts to adjust better the time until crash. At time unit 160 (40 minutes after the start of the experiment), the ratio of memory leaks is reduced to 0-15, so that a crash would be closer if the ratio continued to be that. We can see clearly again how the model detects this fact and adjust (decreases) the predicted time-to-crash again. The next change, 20 minutes later, is also detected. Interestingly, after around 30 more minutes, the algorithm indicates that a change has occurred and starts predicting longer (too long, in fact) times until crash: This is due to the garbage collector activating at that point and freeing up memory in the JVM Heap, which looks to the predictor as if the injection rate is lower than real. After that temporary noise disappears, the model adjusts the time again.

In figure 6 we can see more clearly the error between the real and predicted time until crash. The results are accurate...
enough to help taking informed decisions at least during at least during an important phase of the execution.

2) “Red light” alarm prediction: After these results, we decided to add a new way of increasing the accuracy of our framework because, with numerical prediction, we can see how during few minutes near the crash the prediction is unacceptable. For this reason, we decided to use a classification method to try to predict if the system is in the red zone (5 minutes before the crash) the orange zone (5 to 10 minutes to crash), or the green zone. Using this method in conjunction with the numerical technique we can decide with less error whether action should be taken.

We can observe in Table II that J48 is able to distinguish all the real red or orange zone. The 7 instances predicted as green zone being red zone are due the same problem that suffers M5P when the crash is coming presented before.

TABLE II
CONFUSION MATRIX USING J48.

<table>
<thead>
<tr>
<th></th>
<th>Green Zone</th>
<th>Orange Zone</th>
<th>Red Zone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green Zone</td>
<td>432</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Predicted Values</td>
<td>Orange Zone</td>
<td>20</td>
<td>0 0</td>
</tr>
<tr>
<td>Red Zone</td>
<td>7</td>
<td>0</td>
<td>12</td>
</tr>
</tbody>
</table>

We applied different techniques to find what makes a failing component different from a correct component. The main attributes to check are the ones related to the resource evolution, and the consumption of each component at each time unit. We work under the following hypothesis: While correct components will follow the resource evolution punctually, the failing component will follow, at long term, the abnormal resource use increase. So we propose to determine faulty components by computing some form of distance between the sequences of observed resource consumption (VII-A). We choose the following notion of distance, for reasons that will be explained in the long version of the paper: let $X_{k,i}$ be the normalized resource consumption of the component $k$ in the time $[i-1,i]$, and $Y_i$ the normalized variation of the resource in the same time interval, and the distance is computed as follows:

$$X_{k,i} = \frac{\text{Consumption}_{k,i} - \text{mean}(\text{Consumption}_k)}{\text{StDev}(\text{Consumption}_k)}$$

$$Y_i = \frac{\text{ResourceVar}_i - \text{mean}(\text{ResourceVar})}{\text{StDev}(\text{MemVar})}$$

$$\text{Distance}_k = \frac{\sum_{i=0}^{n}(\sqrt{\|X_{k,i} - Y_i\|})}{n} \quad (4)$$

If the two sequences were exactly the same, their distance would be 0. In general, the component that is most guilty of consuming the resource should have the smallest distance to the one measuring the resource overall.

B. Experiment Description

We now want to check that indeed this method can determine the most suspicious components in a crash caused by a memory leak. Ideally, the method should detect a failing component, independently of the size of the injection, the rate of injection, and having other components also injecting.

For this, we test the following scenarios:

- A single component injecting memory with different order sizes.
- Different components injecting simultaneously.
- Different components injecting simultaneously with different rates of injection each component.

We will measure, for each component, the average consumption of memory for each time unit (approximately 15 seconds), and the corresponding memory variation for each

![Image](Fig. 6. Error between real and predicted time until crash)
time unit, at three levels: system memory, Tomcat memory, and the Tomcat memory heap, where the injection is taking place. We considered six components in the experiment, named A to F. Some of the component call others, and in particular the dependences are (B→A, C→A or D, E→C) among others, where → is read “calls”.

In figure III we can see the results of the method applied to a single component injection. We can see that there is always another component joining the guilty one at the top of the ranking, which is always one that calls the guilty one; in that sense, the calling component becomes an accomplice, but not the true cause of the failure.

In tables IV and V we can see the results for the detection of multiple failing components, also with different rate of injection for each component. We can see that, for these experiments, the guilty components are marked as first suspects or suspects right after their dependent (calling) components. In other words, we do not always single out the truly faulty components, but we can narrow down the list of suspects considerably. This is an enormous advantage over having to analyze all the components in the system, which is absolutely prohibitive.

C. Additional Observations

Given the results of the experiments, a few aspects need additional discussion.

Components are interdependent. If a correct component calls systematically a failing component, it is easy to relate the execution of this component with the crash, even if it is not the directly guilty one. It may be future work to be able to examine dependencies between components and tell apart with more precision the memory consumption actually due to a component from that of the components it calls.

As we can not examine periodically the size of all the components, we have to measure the state of the memory before and after the component execution. This brings a notorious problem when the garbage collector triggers in the middle of a measure. While not being a fatal problem, it creates some noise in the measures that can lead to a few false positives. Several detection methods we tried failed because of precisely this problem, and that brought us to the particular way of ranking components by suspicion, which seems to work properly as we demonstrated in subsection VII-B.

Also, not all components are not executed with the same frequency. This makes harder to distinguish a guilty component that is called less often than other components, just because of statistical fluctuations.

VIII. Conclusions and Future Work

We have presented a complete self-healing framework that predicts the time until crash and the status of every node of the cluster and, when a crash approaches, triggers a clean recovery mechanism. The recovery mechanism avoids missing any ongoing and new request, so the system keeps running from the user’s perspective. We have combined two different machine learning algorithms (M5P and J48) to predict the effect of Heisenbugs that consume resources until its exhaustion, in a way that cannot be attributed to excess load. The combined effect of both algorithm provides us with both a numerical estimate of the time until crash and a green-orange-red status for each cluster depending on the imminence of the crash.

However, several issues and challenges are still open. It is needed to work on the decision system, which has the responsibility to understand the prediction information and take the most effectiveness action to maximize the cluster utilization. And we want to discuss in the future, for instance, introducing some IA (intelligence artificial) to introduce expert decisions according the prediction data offered by the predictors.

On the other hand, we have presented a post-mortem technique to determine the anomalous application component which provokes the crash. We have used an on-line fine-grain monitoring system to collect the resource consumption of the components and an online module to find correlations between the overall consumption rate and the consumption rate of each component. We observed that, as we go deeper into the memory heap structure, we can determine better the cause of the memory leak. While the memory system looks noisy from the outside, the view gets clearer when measuring separately the internal components of the memory heap, so it is interesting to monitor the correct resource and in a correct way. A weakness of our current approach is that we cannot right now always determine the really guilty component due to the coupling between modules and application levels. For this reason we plan to work in a next version that can obtain a component dependency map of components and use this information to allocate faults more precisely.

REFERENCES

**TABLE III**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A injects 1Mb</td>
<td>0.53903</td>
<td>0.63293</td>
<td>0.50337</td>
<td>0.64137</td>
<td>0.60776</td>
<td>0.58227</td>
</tr>
<tr>
<td>D injects 1Mb</td>
<td>0.91309</td>
<td>0.87799</td>
<td>0.90156</td>
<td>0.89463</td>
<td>0.90863</td>
<td>0.93065</td>
</tr>
<tr>
<td>A injects 10kb</td>
<td>0.92138</td>
<td>0.91857</td>
<td>0.95337</td>
<td>0.92656</td>
<td>0.92746</td>
<td>0.97198</td>
</tr>
<tr>
<td>A injects 10kb</td>
<td>0.63800</td>
<td>0.61635</td>
<td>0.72226</td>
<td>0.74611</td>
<td>0.73598</td>
<td>0.85464</td>
</tr>
</tbody>
</table>

**TABLE IV**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A injects 1Mb</td>
<td>0.53903</td>
<td>0.63293</td>
<td>0.50337</td>
<td>0.64137</td>
<td>0.60776</td>
<td>0.58227</td>
</tr>
<tr>
<td>A, D inject 1Mb</td>
<td>0.79260</td>
<td>0.72315</td>
<td>0.69170</td>
<td>0.79746</td>
<td>0.79185</td>
<td>0.83656</td>
</tr>
<tr>
<td>A, B, D inject 1Mb</td>
<td>0.68452</td>
<td>0.75076</td>
<td>0.62173</td>
<td>0.63872</td>
<td>0.70111</td>
<td>0.82340</td>
</tr>
<tr>
<td>A, B, C, D inject 1Mb</td>
<td>0.74444</td>
<td>0.68121</td>
<td>0.68170</td>
<td>0.69473</td>
<td>0.71479</td>
<td>0.79431</td>
</tr>
</tbody>
</table>

**TABLE V**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, D inject 1Mb with different rates</td>
<td>0.75374</td>
<td>0.65428</td>
<td>0.65614</td>
<td>0.58907</td>
<td>0.74566</td>
<td>0.82210</td>
</tr>
<tr>
<td>A, B, C, C inject 1Mb with different rates</td>
<td>0.66854</td>
<td>0.70608</td>
<td>0.56020</td>
<td>0.64900</td>
<td>0.56516</td>
<td>0.85042</td>
</tr>
</tbody>
</table>

[23] Linux Virtual Server: http://linuxvirtualserver.org