Predicting web application crashes using machine learning

Javier Alonso¹,², Josep Ll. Berral¹,³, Ricard Gavaldà³, and Jordi Torres¹,²

¹ Dept. of Computer Architecture, Technical University of Catalonia
  C/Jordi Girona 1-3, Campus Nord, C6 Building
  Barcelona, Spain

² Barcelona Supercomputing Center (BSC)
  C/Jordi Girona 1-3, Campus Nord, C6 Building
  Barcelona, Spain

³ Dept. of Software, Technical University of Catalonia
  C/Jordi Girona 1-3, Campus Nord, Omega Building
  Barcelona, Spain

{alonso, berral, torres}@ac.upc.edu, {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 resource exhaustion for different reasons: overload, inadequate system resource planning, or transient software failures which consume resources until crash. Several previous work have proposed the use of machine learning algorithms for modeling and predicting resource consumption, and the effectiveness of these approaches have been demonstrated in failureless, stationary circumstances. In this paper, we present a framework based on machine learning techniques to predict the time to crash when the system suffers transient software errors such as memory leaks which consume resources randomly and gradually. The experiments illustrate that our approach is effective at predicting crashes (at least of a particular type, those due to memory leak) and with a lot of potential impact. Moreover, we present a discussion about the monitoring systems granularity level.

1 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, and only autonomic computing seems to be the solution[1].

Because system complexity is growing day by day, the number of failures due (directly or indirectly) to this complexity has also been growing, resulting in undesirable
behaviors, poor levels of service, or even total outages. The need to prevent or grace-
fully deal with outages of business and critical systems is clear considering attention of
the huge loss due the downtime per hour for the industry, as reported in [2][3]. More-
over, outages have a negative impact on the company image that could affect indirectly
profits. For this reason, building high availability servers has become a hot topic in the
last years.

Clustering [4] is possibly the most used technique to minimize outages in indus-
try, and today most business-critical servers apply some sort of server-redundancy,
load-balancers, and fail-over techniques. This is certainly a solution tolerate applica-
tion crashes, but has the inconvenient of additional hardware cost. A more recent trend
complements redundancy with self-healing techniques [1], which help automate recov-
ery procedures and prevent the occurrence of unplanned failures. Also, virtualization
technologies are been used by the industry to deploy clustering techniques reducing the
total cost of ownership (TCO) of the systems [5].

The literature divides the reasons for downtime in three main categories: Human or
operator errors, software errors, and hardware errors. According to [10], the distribution
of errors in these categories was (approximately) 40%, 40% and 20%, respectively, by
2005. If we observe in detail the evolution of this distribution from 1985 to 2005 [6–
10], we can see that the proportion of hardware errors has decreased (from 32% in the
80s to 15-20% by now). The number of operator errors remains fairly stable along time:
Although system complexity has grown and keeps growing, and this would suggest an
increase in human errors during the management tasks, the operators currently have
better administration tools to automate some parts of their task. Nevertheless this 40%
shows clearly that there is important room for improvement.

Also, according to [6], the software faults which causes software errors can be di-
vided in two main categories: Heisenbugs and Bohrbugs. Bohrbugs are deterministic
and (relatively) easy to fix using traditional testing or debugging techniques by devel-
opers. Heisenbugs, on the other hand, manifest themselves and disappear nondetermi-
istically, 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 [11]. 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 or
a set of them.

Moreover, software errors have been growing up along the time (from 25% in 80s
to 40% today), due mainly to the complexity of the current software and the heteroge-
neous environment where our systems have to work. According to [6, 7, 9, 10], the most
complex failures to overcome are the transient or intermittent failures. These transient
or intermittent failures are caused by dormant faults (Heisenbugs) that, under determi-
nate circumstances, wake up and temporary disappear after a while. The Heisenbugs
which cause exhaustion of some resource (memory, CPU time, processes, connectivity,
etc.) resulting on undesirable behavior of the system or even hang ups or crashes are the
most complex faults to predict or model.
Resource exhaustion could be caused by different reasons, and some of them cannot be considered anomalies or faults. For example, system overload: the system works correctly, but it does not have enough resources to deal with the workload it receives. These are relatively easy to predict, because the one can match the relevant workload features with the system’s available resources. On the other hand, transient software failures are more difficult to deal with, because they often depend on unforeseen interactions among components, appear under unusual logical conditions, and are hard to reproduce in isolation so that developers can fix them. But they are important: if transient failures happen keep consuming resources (such as memory or CPU) they invariably make the system unstable and lead to performance degradation or crash. We have to take into account that transient software failures are due to permanent faults, however these permanent faults are transient in nature, because sometimes they are dormant.

Machine learning algorithms have been used to predict system behavior under normal circumstances (no errors) to estimate resource usage under varying workloads, and for capacity planning. There are, however, fewer studies in using them to indeterministic degradation and crashes prediction. On the other hand, the machine learning algorithms have been used to anomaly detection more than to predict the time until crash.

In this paper we propose a framework based on machine learning algorithms to predict the time until crash for web application servers due to indeterministic software faults. We have evaluated different off-the-shelf algorithms to determine their effectiveness in different circumstances and workloads, to predict (numerically) the time until crash. In [30] we presented a preliminary study of the two families of machine learning algorithms: Linear Regression and Decision trees. However, in this paper we conduct a more detailed analysis and changed the accuracy evaluation approach: we evaluated the algorithms according their accuracy to predict the time until crash as the crash is near. Moreover we have introduced a prediction system to flag “red-light” alarms when a crash is imminent. Although so far we have run our simulations off-line, our ultimate goal is to implement an on-line, real-time prediction system. For this reason, we have concentrated algorithms with low computation needs. Moreover, after conducting our experiments, we present a preliminary discussion about the grain level of the monitoring systems to be useful to predict/model the behavior of the systems. On the other hand, we want to notice that a companion paper has been submitted elsewhere where we present a complementary work [31]. In this paper, we used the same theoretical framework and similar machine learning techniques. However, in the current paper we pay more attention on the machine learning algorithm details and the monitoring grain level, while in [31] we focused on the on-line prediction and recovery framework and root cause determination.

The rest of the paper is organized as follows: Section 2 presents the related work in the area. Section 3 presents in detail our mathematical model and our prediction strategy. Section 4 describes the details of our framework. Section 5 describes the experimental environment used to execute the experiments and the results obtained. Section 6 presents a preliminary discussion about the monitoring systems grain level, and Section 7 presents some conclusion and future work.
2 Related work

The idea of predicting when a system will exhaust a given resource is not new. A lot of works have modeled this behavior using different machine learning and analytic approaches with successful results. Works such as [12, 13] 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 [12], 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 [13], authors use time-series ARMA models from the system data to estimate the resource exhaustion due to workload received by the system.

In [19] presents an off-line framework to develop performance analysis and post-mortem analysis about the causes of Service Level Objective (SLO) violations. They propose to use TANs (Tree Augmented Naive Bayesian Networks), a simplified version of Bayesian Networks (BN), to determine which resources are most correlated with the performance behavior.

In [18], 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.

In [17], authors present an on-line framework that allows to determinate if the system is suffering an anomaly, workload change or 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 the two resulting segments there is some model with acceptable error, they determine that we are in front of a software update. If for a performance period it is impossible to obtain a sequence of linear regressions with acceptable errors, the conclusion is that the system is suffering some type of anomaly during that period.

The Pinpoint project [21] 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 [20] collects resource consumptions by components to model with high accuracy the behavior of the system, even distributed ones. However, Magpie and Pinpoint need to instrument the application server. In our approach, we get a less intrusive approach that still guarantees flexibility and adaptability.

Furthermore, concerning failure and critical event prediction, several studies have been conducted in other areas such as Telecommunications [14–16]. However, these prediction techniques are not sufficient to manage the complex and varied system health data in computer systems.

In the preceding paper [30], we presented a preliminary study of which prediction methods were adequate for predicting the time until crash under a random error which consumes memory, and concluded that methods combining decision trees and linear regressions seemed to give much better results than either decision trees or linear regression alone. Now we take those conclusions for granted, but here we: 1) present our underlying mathematical model and assumptions in detail, 2) present an experimental setup that can test performance in dynamically varying the memory leak injection
rate (as opposed to testing under a single workload with fixed memory variation injection), and 3) concentrate on the problem of detecting approaching and imminent crashes (“orange alerts” and “red alerts”) rather than trying to produce accurate time-to-failure estimates long before failures happen. The alarm-detecting is the crucial element to apply the self-healing or demand the human operators’ attention. The level of depth in the monitoring system of the health state is also discussed here.

3 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.,

$$R_{i,t} = R_{i,t-1} + \delta_{i,t}$$  \hspace{1cm} (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 stationarity 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 \( W \), 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 use 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 \( T_{\text{fail}} > t \) such that \( R_{i,T_{\text{fail}}} \) 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

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

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

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

Under this constant-degradation-speed hypothesis, one should therefore make the prediction that failure will occur around time \( T_{\text{fail}} \). 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 \( T_{\text{fail}} \)”; 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 \( R_{\text{max}} \). \( \delta_i \) can be estimated as the average of most recent observations, with some smoothing (see next paragraph). \( R_{\text{max}} \) 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 [32], 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.

4 The Framework

We have developed a new framework designed towards the ultimate target of our research, that is, an on-line and real-time system for predicting time until failure and approaching crashes. We are, however, at a preliminarily stage testing our method and framework in off-line environment. As long as the instrumentation and prediction methods we use are computationally light, we do not foresee major obstacles in transporting our prototype to the on-line setting.

We can divide our framework in two phases: the training phase and the testing or prediction phase. The training phase has the responsibility to generate the model or classifier. The testing or prediction phase applies this model to the current real-time observations to determine whether the system is approaching a crash. In figure 1, we present an overview diagram of the framework architecture. While this figure presents the whole system, the white module (Action Recovery Manager) is out of scope of this paper because it will be too closely dependent on the system where it is deployed. We will focus on the dark gray modules, the ones that collect and process the information to be handled by the Action Recovery Manager.

![Fig. 1. Detailed Architecture of the Prediction/Action Framework](image)

The Monitoring Agent has the responsibility to collect system metrics from the system. The metrics that we have decided to collect are a small set easily obtained from any O.S.: throughput, response time, workload, system load, disc usage, swap used, number of processes, number of threads, free system memory, memory occupied by tomcat, number of http connections received and number of connections to the data base. To collect all of this information we have modified Nagios [22], a well-known and used by the industry monitoring tool that allows to execute monitoring gadgets at the system under monitoring every n seconds (in our case will be 15 seconds) and store the results in a file or database.
The typical usage in this offline mode would be the following: We collect all data from several system executions, possibly under different workloads, using the monitoring tool. After that, with the aid of an expert, we locate where in the data set crashes have occurred (for example, which values of response time or throughput are unacceptable). The new data set is used as input for the Enriching Process. The Enriching Process main task is to add additional, derived, variables to the tuples of system observations. In particular, in the current version of this module, the most important derived variables we introduce are the EWMA-smoothed out values of all measured resources and of their evolution along time. The idea is to be able to detect potential changes of the consumption of any resource. Most importantly, for training purposes, the Enriching Process adds an extra value with the time-to-failure for each tuple of observations; note that this requires looking forward in the dataset to find the next tuple which the expert indicated as crash.

After enriching the data, it is used to build a model, and this model is tested with new data sets or using different data mining techniques like percent split [27]. The flexibility of our framework allows to add new machine learning algorithms for model building easily, or even to have several algorithms and create a committee of models for increasing the accuracy of the predictions. This latter idea was used, e.g., in [29], where they have four models and choose the result by general consent (at least 3 out of 4 must agree). Figure 2 presents the activity diagram between the different modules presented before.

![Activity Diagram](image)

**Fig. 2.** Detailed behavior of the Prediction/Action Framework

## 5 Test Experiments

### 5.1 Experimental Setup

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 [24]. We have used the Java version developed using servlets and using Mysql [25] as database server.
As application server we have used Apache Tomcat [26]. 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. The details about the machines characteristics are depicted in Table 1.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Clients</th>
<th>Application Server</th>
<th>Database server</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2-way Intel XEON</td>
<td>4-way Intel XEON</td>
<td>2-way Intel XEON</td>
</tr>
<tr>
<td></td>
<td>2.4 GHz with 2 GB</td>
<td>1.4 GHz with 2 GB</td>
<td>2.4 GHz with 2 GB</td>
</tr>
<tr>
<td>RAM</td>
<td>RAM</td>
<td>RAM</td>
<td>RAM</td>
</tr>
<tr>
<td>Operating System</td>
<td>Linux 2.6.8-3-686</td>
<td>Linux 2.6.15</td>
<td>Linux 2.6.8-2-686</td>
</tr>
<tr>
<td>JVM</td>
<td>jdk1.5 with 1024MB</td>
<td>jdk1.5 with 1024MB</td>
<td></td>
</tr>
<tr>
<td>Software</td>
<td>TPC-W Clients</td>
<td>Tomcat 5.5.26</td>
<td>MySql 5.0.32</td>
</tr>
</tbody>
</table>

**Table 1. Detailed Machine Description**

TPC-W clients, called Emulated Browsers (EBs), access the web site (simulating an online 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 (uses a negative exponential distribution with 7 seconds), 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 a servlet (*TPCW_search_request_servlet* class) of the TPC-W implementation. This 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 on 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. TPC-W has three types of workload (browsing, Shopping and Ordering). In our case, we have conducted all of our experiments using shopping distribution. The EBs in this workload visit the servlet modified around 20%. This makes that with high workload our servlet injects quickly memory leaks, however with low workload, the consumption is lower too. The memory leak injected in our experiments is fixed (1MB) in all experiments. We could have made this size random 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.
5.2 Experimental Results

Model Evaluation under a Fixed Environment  In our previous work [30], we evaluated two families of algorithms: linear regression and decision trees, and we concluded that decision trees are in general better options because linear regression is extremely sensitive to outliers. Outliers are very common when the system is suffering transient errors, and so linear regression obtained poor results. We further identified the M5P algorithm, which combines linear regressions with decision tree, as superior to using either technique alone. Also we measured the accuracy using the Mean Average Error (MAE) over all the predictions. However MAE is not as important as having low error near the crash, the point where we want to be the most accurate as possible, so we re-evaluated the method focusing on the last minutes before the crash (we chose the last 10 minutes as critical period).

We repeated the experiments presented in [30]: We injected a memory leak of 1MB on \( N/2 = 15 \) visits to the faulty servlet in average, with constant workload (recall that we fixed \( N = 30 \)), and Tomcat’s heap size set to 1GB. The model was trained using different workloads (number of EB’s) all under this memory leak scheme. We merged the five logs resulting of using 5, 25, 50, 100, and 200 Emulated Browsers (EBs) into a single training data set. After that, we evaluated the model under two different workloads never used to the training process, 75 and 150EB’s but, this time, unlike [30], we focused on the accuracy during the last 10 minutes before the crash.

Figure 3 presents the comparison of the three studied models (linear regression, RepTree and M5P) when tested against the test set obtained with 75EBs as workload. We present the error evolution obtained in front of real time until crash (zero value is when the algorithm predicts exactly the time until failure). We can see how the M5P and RepTree have a better average error than Linear Regression. However if we zoom in on the last 10 minutes of the experiment, showed in figure 4, we can see how the RepTree is closer to the real time until failure, and even has an error of about 200 seconds. This result contrasts with our first analysis because we evaluated the average error over several hours: We believe that the linear regressions contained in M5P, which
are reasonably accurate during stable phases, suffers too much from the fluctuations occurring near the crash, while RepTree, which makes fixed decisions at the leaves rather than linear regressions, is more immune to such fluctuations.

In table 2, we present the MAE comparison between the three algorithms. We can observe as Decision Tree algorithms (M5P and RepTree) obtains better results when we analyze the complete experiment, however, if we focus on the critical zone (fixed by us in the last 10 minutes) the results are quite different. RepTree becomes the best approach during this phase, as can be observed from table 2.

<table>
<thead>
<tr>
<th></th>
<th>M5P</th>
<th>Linear Regression</th>
<th>RepTree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete experiment</td>
<td>1946.42</td>
<td>4077.97 sec</td>
<td>1390.84 sec</td>
</tr>
<tr>
<td>with 75EBs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last 10 min</td>
<td>1403.75</td>
<td>422.82 sec</td>
<td>354.09 sec</td>
</tr>
<tr>
<td>experiment with 75EBs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete experiment</td>
<td>920.21</td>
<td>3446.99 sec</td>
<td>913.52 sec</td>
</tr>
<tr>
<td>with 150EBs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last 10 min</td>
<td>1386.19</td>
<td>443.00 sec</td>
<td>359.44 sec</td>
</tr>
<tr>
<td>experiment with 150EBs</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Mean Average Error for 75EBs and 150EBs during the experiments, focusing on the last 10 minutes before the crash

Another interesting result was that linear regression was pessimistic in its predictions and predicted the crash far too early due to the outliers that the transient failure creates on the system behavior. For more details on the evaluation under this fixed environment, see [30].

Detecting Imminent Crashes under a Fixed Environment  After this focused evaluation, we concluded that the best strategy to predict the time until failure would be to have different models depending on the state of the system. If our system is far from crashing, M5P seems like a good option, however, if we are near the crash, M5P is a bad option and RepTree seems better. Because of this, and in itself, it would be interesting
to simply distinguish whether or not a crash is approaching. This called for classifier algorithms, whose output is not a number but one among a binary, ternary, etc. set.

To allow for some flexibility, we distinguished three rather than two periods: Tuples in the dataset were labeled Red, Orange, or Green. Tuples in the last 10 minutes before the crash were labeled Red, those in the 10 minutes before the Red zone were labeled Orange, and all others (i.e., at least 20 minutes before the crash) were labeled Green. We still are primarily interested in distinguishing Red from non-Red, but we will not count Red tuples classified as orange as very severe mistakes. Red tuples classified as Green are probably the most expensive mistake, as they mean an unpredicted crash. Green tuples classified as Red are false positives: they could mean starting preventive measures before they are strictly necessary or even without being necessary at all, which may be a nuisance but not as much as a crash.

Among the many classifiers in the literature, we took three incorporated into the WEKA package [27]: J48, an implementation of the well-known C4.5 decision tree inducer, NaiveBayes, a probabilistic classifier based on Bayes’ rule, and IBk, the nearest neighbor classifier. We used WEKA’s default options in all three cases. We remark our use of off-the-shelf classifiers to indicate that we included hardly no domain knowledge into the prediction system and that, obviously, there is ample room for improvement by designing ad-hoc algorithms. The reason for choosing these three is that they are computationally light (with Naive Bayes being the cheapest and IBk the most expensive of the three).

<table>
<thead>
<tr>
<th></th>
<th>Real Value</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Green Zone</td>
<td>Orange Zone</td>
<td>Red Zone</td>
</tr>
<tr>
<td>Predicted Values</td>
<td>Green Zone</td>
<td>207/210</td>
<td>0/10</td>
</tr>
<tr>
<td></td>
<td>Orange Zone</td>
<td>245/0</td>
<td>4/10</td>
</tr>
<tr>
<td></td>
<td>Red Zone</td>
<td>2/0</td>
<td>16/0</td>
</tr>
</tbody>
</table>

Table 3. Confusion Matrix for J48. X/Y, with X for 75EBs and Y for 150EBs

The results are shown in the form of confusion matrices, indicating how many examples of each class (red, orange, green) are classified as in each class. These matrices provide better information than simply the number of errors. Perfect predictions are those in the diagonal. E.g., red-as-orange confusions are wrong, but intuitively not as wrong as Red-as-green.

Table 3 presents the results obtained using J48 under the two test data sets. We can observe that J48 classifies many Greens as Oranges, but makes none or very few “severe” Green-Red mistakes.

Table 4 presents the results obtained using Naive Bayes under the two test data sets. We can observe that Naive Bayes model also makes few Green-Red mistakes, but tends to place a good number of both Reds and Greens in the Orange zone.

Table 5 presents the results obtained using IBk under the two test data sets. We can observe that IBk model has an acceptable accuracy for all zones, however has mistakes in all of them. In particular, they do classify 6 Red tuples as Green in the 150 EBs dataset.
The next step, not yet carried out, would be to combine the different predictors (numerical and classifier) in a wise advisory board to predict the crash of the system and the time until failure with acceptable level of accuracy. In particular, different classifiers would be trusted in different types of system states.

**Model Evaluation under a Dynamic Environment** So far, we have evaluated the accuracy of different set of models to predict crashes under a fixed workload (though trained with different workloads). We now move one step ahead and evaluate our models in a dynamic environment, i.e., one in which the workload or error conditions vary over time.

We conducted a new experiment where the rate of injection of memory leaks is changing over time. During the first 30 minutes we do not inject any memory leak, so the memory consumption is the normal behavior of Tomcat under normal workload. After the first 30 minutes, we start injecting memory leaks following the distribution of one injection between 0 to 30 visits to the servlet. After 30 more minutes, we started a more aggressive memory leak (injection between 0 to 15 visits) and finally, after another 30 minutes we relaxed the memory leak to one injection between 0 to 75 visits to the servlet. The idea is to test our predictor’s ability to detect the different rates and answer appropriately. That is, the prediction at any given time should be taken as “if memory continues to shrink at the current rhythm, we can expect crash by time X”.

To compare the new environment with respect to the previous one, Figure 5 presents the (EWMA-)smoothed memory consumption rate during two experiments using the 100EBs workload. In the new environment we can see how the fault injection rate is changing in time becoming far less deterministic, which in principle should make predictions harder. Furthermore, we can observe flat zones when the Tomcat memory achieves the 50% and the 85%. These flat zones indicate that seems that the Tomcat memory remains stable during that zones, indicating a 0 memory speed consumption, but this is not the right explanation, as we will present in section 6.
The new environment changes the rate of memory injection every 30 minutes. During the first 30 minutes no failure is injected, so memory remains constant and the predicted time to failure should be “infinite”. We used 99,999 in the time-to-failure column of the dataset meaning “infinite”, and this value is visible in the graph.

Figure 6 presents the prediction error between predicted time-to-crash and real time-to-crash using linear regression. The x-axis has been reduced to clarify the graph because linear regression produces huge numbers when no failure is in sight (i.e., to mean “infinite time”). Even excluding this pathological figures, the error of linear regression is very high in average. Figure 9 shows the last 10 minutes of the experiment and again we have modified the x-axis maximum to help understanding the graph. We observe how the linear regression obtains a poor accuracy even in this range. We conducted the same experiment under the model generated using RepTree. Figure 7 presents the results. We observe that the results are good again. The RepTree algorithm is able to
model the behavior under transient memory leaks, even if injected at different speeds during the experiment.

Figure 8 presents the results obtained using the model created by the M5P algorithm. This Model has an acceptable level of accuracy in average but, as already presented in Figure 9, this model loses accuracy when the system is approaching the crash.

Finally, we evaluated the accuracy to predict the crash within the last 10 minutes before the crash. Figure 9 presents the results. As in the case of a constant workload discussed before, RepTree obtains the best results. We can observe the MAE from the dynamic experiment with more detail in table 6.

**Detecting Imminent Crashes in a Dynamic Environment** Finally, we considered whether, under a changing workload, one can distinguish Red from Green zones. Again,
**Fig. 9.** Comparative detail last 10 minutes of experiment. Workload 100 EBs and dynamic environment

<table>
<thead>
<tr>
<th></th>
<th>M5P</th>
<th>Linear Regression</th>
<th>RepTree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete dynamic experiment</td>
<td>15238.41 secs</td>
<td>28608.04 secs</td>
<td>4183.52 secs</td>
</tr>
<tr>
<td>Last 10 min dynamic experiment</td>
<td>655.79 secs</td>
<td>50193.07 secs</td>
<td>180.18 secs</td>
</tr>
</tbody>
</table>

**Table 6.** Mean Average Error for dynamic experiment, focusing on the last 10 minutes before the crash

We used the J48, Naive Bayes, and IBk classifiers, and results are shown in tables 7, 8, and 9 respectively.

<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>388</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Predicted Values</td>
<td>Orange Zone</td>
<td>83</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Red Zone</td>
<td>25</td>
<td>19</td>
</tr>
</tbody>
</table>

**Table 7.** Confusion Matrix using J48.

We can observe that J48 is able to distinguish all right whether the system is green or red zone. Interestingly, it places all instances from the Orange zone in the Red zone. For Naive Bayes the results are worse than J48: it tends to place Red zone instances in the Orange zone, which is not terribly bad but not as satisfying. Finally, IBk obtains worst results than the others: it places 13 out of 18 Red instances in the Green zone. This prediction is too dangerous for our system under monitoring because it means imminent crash with no corrective measures taken.
Table 8. Confusion Matrix using NaiveBayes.

<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>369</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Predicted Values</td>
<td>Orange Zone</td>
<td>113</td>
<td>0</td>
</tr>
<tr>
<td>Red Zone</td>
<td>14</td>
<td>14</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 9. Confusion Matrix using IBk.

<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>479</td>
<td>18</td>
<td>13</td>
</tr>
<tr>
<td>Predicted Values</td>
<td>Orange Zone</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Red Zone</td>
<td>14</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

6 New Evaluation using a fine-grain monitoring system

In this section we introduce a discussion about the depth level of the monitoring systems. As we presented before in figure 5, the Tomcat memory variation during the execution with a constant memory injection ratio had two flat zones when the Tomcat memory was around 50% and 85% (the same happens in dynamic experiment, but the flats are less obvious), provoking a zero speed memory consumption during these zones. However, we still performed memory injections during this time, and thus we have to ask what is the reason for these flat zones. After some investigation, we found the explanation in a particular feature of the Garbage Collector in this particular application, which requires some detailed explanation.

The monitoring system collects the metrics from the operating system, and the operating system used in our experiments was Linux. The Linux memory management system doesn’t recover the memory released by applications until the operating system needs that free but cached memory. For this reason, we decided to add a new plug-in to the monitoring system to collect data information from the Heap Memory, inside the Java Virtual Machine (JVM). The Heap Memory is divided in three areas: Young, Old and Permanent (our memory leak injections are done into the Old zone). In figure 10, we can observe how these three sections are evolving during the experiment with a 100 EBs workload injecting 1MB using N=30. So then we can see the real behavior of the Tomcat memory. Although, we have configured our Tomcat with 1GB of maximum memory and a starting memory allocation of 512MB. When the starting memory becomes full, the Heap resizes the Old section to allow the Tomcat application server continue running. During this resizing phase the Garbage Collector (GC) is freeing memory. However, from the operating system point of view, the Tomcat memory consumption is zero, which is not true.

Based on this experiment, we can conclude the importance to monitor the resources in detail, even the virtualized resources, to understand and model the behavior of the systems. If we only monitor the operating system metrics we can loose important and critical data to predict that behavior more accurately. Another interesting point is that
Heap Memory is a virtualized memory managed by the JVM. Currently, virtualization is becoming a hot topic to build cloud computing systems. If we want to model the behavior of these new complex and virtualized environments, a new family of monitoring systems is needed, allowing the administrators to know the real situation of the physical and virtual resources. Traditional physical resource monitoring systems are not enough, so we need to monitor the virtualized resources to know if one virtual machine is exhausting its virtual resources and predict potential anomalies or overloads, resulting in undesirable behaviors or even hang ups or crashes. Some works like [34] have been started in order to monitor the resources consumed by virtualized applications, showing the importance of being able to use a more fine-grain monitoring systems to know the real state the physical resources, but also the virtual resources.

Based on the Heap Memory behavior, we decided to monitor the Heap in detail (metrics added: Young memory used, Young maximum memory available, Old memory used, Old maximum memory available, Permanent memory used, and Permanent maximum memory available), improving the results, as shown in figure 11, where we can see the error obtained using M5P in the dynamic environment presented before. M5P is the best option using the fine-grain resource monitoring system, allowing understand better what is the behavior of the memory.

7 Conclusions and future work

We have shown that, for one particular type of transient error (intermittent memory leaks) it is possible to predict with reasonable accuracy the time remaining for a crash and whether or not a crash is imminent. This opens the possibility of combining prediction with self-healing techniques to avoid crashes or malfunctions, which can be extremely expensive or damaging. Moreover, we have presented the need to obtain fine-grain monitoring systems to allow to understand in detail the behavior of the resources, because sometimes, if we use a more general information, we can loose valuable information from the resources.
Among the main lines for future work are: Turning our offline system into a truly online, real-time one. Trying a wider variety of prediction and classification methods, and possibly developing ad-hoc ones, for improved accuracy. Combining several models (obtained by one, or several, learning algorithms) also to improve accuracy. Finally, testing our approach on a real system, and combining it with actual self-healing techniques (i.e., implementing the Action Recovery Manager module from Figure 2), like [33]. Also, we want to measure in detail the impact on the system under monitoring performance to evaluate the effectiveness of our approach.

References

31. J. Alonso, J. Ll. Berral, R. Gavalda, and J. Torres. *Crash prediction and suspicious component determination for proactive recovery management in clustered web applications*. Submitted to revision. (Temporary could be consulted for review purpose in [http://personals.ac.upc.edu/alonso/Research.html](http://personals.ac.upc.edu/alonso/Research.html)).