Machine Learning for Achieving Self-* Properties and Seamless Execution of Applications in the Cloud

Pierangelo Di Sanzo
disanzo@dis.uniroma1.it
DIAG – Sapienza, University of Rome

Alessandro Pellegrini
pellegrini@dis.uniroma1.it
DIAG – Sapienza, University of Rome

Dimiter R. Avresky
autonomic@irianc.com
IRIANC – Munich, Germany

***Abstract*—Software anomalies are recognized as a major problem affecting the performance and availability of many computer systems. Accumulation of anomalies of different nature, such as memory leaks and unterminated threads, may lead the system to both fail or work with suboptimal performance levels. This problem particularly affects web servers, where hosted applications are typically intended to continuously run, thus incrementing the probability, therefore the associated effects, of accumulation of anomalies. Given the unpredictability of occurrence of anomalies, continuous system monitoring would be required to detect possible system failures and/or excessive performance degradation in order to timely start some recovering procedure. In this paper, we present a Machine Learning-based framework for proactive management of client-server applications in the cloud. Through optimized Machine Learning models and continually measuring system features, the framework predicts the remaining time to the occurrence of some unexpected event (system failure, service level agreement violation, etc.) of a virtual machine hosting a server instance of the application. The framework is able to manage virtual machines in the presence of different types anomalies and with different anomaly occurrence patterns. We show the effectiveness of the proposed solution by presenting results of a set of experiments we carried out in the context of a real world-inspired scenario.**

I. Introduction

Performance and availability of computer systems can be affected by the accumulation of anomalies of different nature, such as memory leaks, unterminated threads, unreleased locks, and/or file fragmentation. Most of the times, occurrences of anomalies are complex to predict. Further, discovering the related causes and eliminating errors via software debugging may be hard. In [1], it has been shown that in web applications an average of 40% of anomalies is due to software errors. As a consequence, these kinds of problems need to be addressed by run-time management of systems. This would require continuous monitoring of a system to detect and manage unexpected behaviours, such as excessive performance degradation or service outage. In many cases (such as 24/7 web applications), the ability to timely recovery the system from failures, and/or to restore adequate performance level, is of paramount importance.

An approach to cope with the problem of accumulation of anomalies consists of periodically performing *recovery* actions forcing the system to a "clean" state (namely, a state where the system works without—or with a reduced number of—anomalies). This prevents the occurrence of further anomalies from (quickly) leading the system to unexpected behaviours. An example of recovery action is the so-called *software rejuvenation* [2], which is typically performed by restarting the application responsible of generating anomalies, or even the hosting machine.

A recovery action can be performed at pre-established time instants (*time-based* approach), or *proactively*, i.e. when the system is predicted to approach an undesired state. In the first case, actions are performed independently of the actual working state of the system. Conversely, in the second case, they are performed based on run-time system monitoring. A proactive approach requires more complex techniques than the time-based counterpart. In fact, estimating the best-suited time for recovering the system from failures, or to restore adequate performance level, can be non-trivial, given that predicting the occurrence of anomalies and their effects on the system is typically hard. On the other hand, benefits of proactive management can be significant, in terms of both system service downtime and system performance.

Recently, in [3], it has been shown that the proactive management of software anomalies can efficiently exploit Machine Learning (ML) algorithms to predict the time to crash of applications. In the proposed approach, the system has been trained until crashing in the presence of anomalies, and values of some system feature have been collected (CPU utilization, memory usage, etc ...) After having been collected, values of system features are fed to the a ML algorithm for building a model to predict the Remaining Time to Crash (RTTC) of the system. The benefits of this approach are related to the fact that a recovery action can be executed before the (predicted) failure time, or even before that the system performance goes below a given level. As well, actions could take place before some Service-Level Agreement (SLA) levels are violated, e.g. the response time increases over a given threshold, or availability is below a certain percentage.

In [4], we showed that ML-based approaches can be used to predict the occurrence of both system crashes and other events, such as violations of performance thresholds, also in the presence of different kinds of software anomalies.

In this paper, we present a ML-based framework for Proactive Client-server Application Management (PCAM) in the cloud. PCAM exploits failure prediction models produced by a ML framework that we presented in [4], called F^{2}PM. PCAM targets a client-server application model, with replicated server instances. We assume that server instances are deployed on virtual machines (VMs) provided by a cloud IaaS (Infrastructure as a Service) [5]. Server instances can be added/removed at run-time (by adding/removing VMs), in order to dynamically scale the server pool according to the system workload. Also, server instances are subject to

software anomalies, which can lead VMs to fail, as well as to cause degraded performance level over time. PCAM is able to trigger recovery actions for any VM on the basis of run-time predictions of the its Remaining Time to Failure (RTTF) and on VM performance measurements. The RTTF of a VM is predicted based on the time when a given *failure* condition is expected to be true. The failure condition corresponds to a system crash or to the violation of some user-defined thresholds related to system performance requirements (e.g. when the average response time exceeds a user-defined value). PCAM uses RTTF prediction models generated according to the scheme used in F^{2}PM.

Particularly, with respect to previous literature studies (as we discuss in more detail in Section II), our approach advances in several directions: i) we target more complex and largely common application deployments, where multiple server instances are replicated on different hosting machines, and where different kinds of anomalies can occur, also with different occurrence patterns; ii) we use both ML-based predictions and run-time system performance measurements to decide when a recovery action has to be executed; iii) we evaluate our framework in the case of a test-bed application reproducing a real-world scenario, where we run the system with different configurations and by injecting different anomaly patterns.

The PCAM approach, on the one hand, aims at improving both the system availability (by reducing the system downtime due to failures) and the system performance (by keeping it up to user-established levels). On the other hand, it allows to reduce significantly the management effort for keeping the system operational, supporting *self-* (healing, optimizing, configuring)* system properties . PCAM is not bound to specific applications, because it only requires to monitor parameters at hosting machine and operating system level. Therefore, besides our target experimental web applications, PCAM can be used, as well, in other kinds of client-server applications, acting in a completely *application-agnostic way*.

The remainder of this paper is organized as follows. In Section II we discuss related work. The architecture of PCAM is presented in Section III. Section IV presents experimental data to assess the validity of our proposal. In Section IV-C we highlight the achievements by this paper.

II. Related Work

In [6], the authors propose a proactive prediction and control system for large clusters. The proposal relies on logs containing six types of events categorized into classes (e.g. the availability of specific systems, or performance violation thresholds) and collected during one year of activity of a large (350 nodes) system. By using time series, rule-based classification, and Bayesian networks, the authors filter the initial data, selecting only the entries, which are useful to carry on a prediction. Essentially, as opposed to the above-mentioned work, in this paper we present a framework, which is able to autonomously rejuvenate systems in a complete application-agnostic way, relying on a differentiated set of prediction algorithms, generated by using different ML methods.

In [7], it has been shown that virtualization can be used to implement effective software rejuvenation approaches. The authors address the case of an application server. It is replicated in two VMs. One VM is active and serves all incoming requests, while the other one is in standby mode. When the active machine needs to be rejuvenated, incoming requests are forwarded to the other VM, which becomes active. The active VM is rejuvenated when the application performance goes down a given threshold. The authors present an experimental study, where they evaluated the overall system availability, showing that user can perceive zero downtime time. Further, they show that the overhead introduced by using a VM is about 14% with respect the case of a real machine. Our framework also targets virtualized environments. However, we focus on orthogonal problems with respect to the study presented in [7]. Indeed, we aim at designing and evaluating a framework based on proactive self-rejuvenation for ensuring high availability and improving the system performance.

Proactive rejuvenation in the case of a virtualization-based framework has been studied in [3]. In the proposed solution, rejuvenation of a VM is executed on basis of predictions of the Remaining Time to Failure (RTTF) derived from run-time measurements of a number of system features. Predictions are based on a linear regression model, where Lasso Regularization is used to reduce the number of system parameters to be monitored. The proposed approach has been evaluated in the case of (artificially-injected) memory leaks, assuming that the RTTF is the remaining time for the system to reach a state where the virtual memory is exhausted. Differently from the approach in [3], PCAM can rely on a differentiated set of ML algorithms. Additionally, PCAM—by its reliance on F^{2}PM—can easily address both a differentiated set of anomalies and user-defined decision rules to identify the failure point, e.g. to account for specific SLA levels. Furthermore, in our present proposal we consider a set of distributed VMs in the Cloud, while in [3] only a couple of locally-hosted VMs is considered.

III. The PCAM Framework

In this section, we provide a detailed description of the PCAM Framework. As we discussed in Section I, we assume a client-server application model, where client requests are served by a set of replicated servers deployed on virtual machines (VMs) of a cloud IaaS. In our implementation, when a VM is predicting to approach the failure condition, PCAM exploits software rejuvenation to restore a proper working state of the VM. This is simply achieved by restarting the VM. However, we remark that PCAM can be used to trigger any other kind of recovery action, which can be customized by the user.

In the rest of this section, we first describe the architecture of PCAM, then we describe the ML-based approach to predict RTTF of VMs and we discuss the on-line control loop performed by PCAM.

A. Framework Architecture

The PCAM architecture (Figure 1) includes a VM acting as a controller (VMC) and k couples of VMs (slave VMs) acting as (replicated) servers. The slave VMs of a couple c_x (with $x \in [0, k-1]$) are named $VM1_x$ and $VM2_x$, respectively. VMC and the slave VMs communicate via message exchange.

On the VMC side, the following components of PCAM are installed:

- A Communication Unit (CU), which is in charge of communicating with all slave VMs;
- A Prediction Unit (PU), which provides RTTF predictions of slave VMs;
- A Load Balancing Unit (LBU), which forwards (remote) clients' requests to slave VMs;

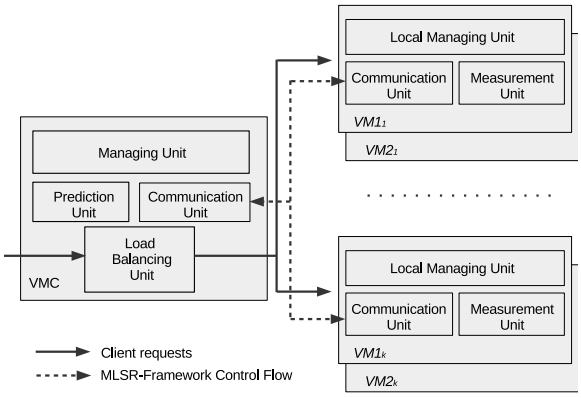

Fig. 1. The PCAM Architecture.

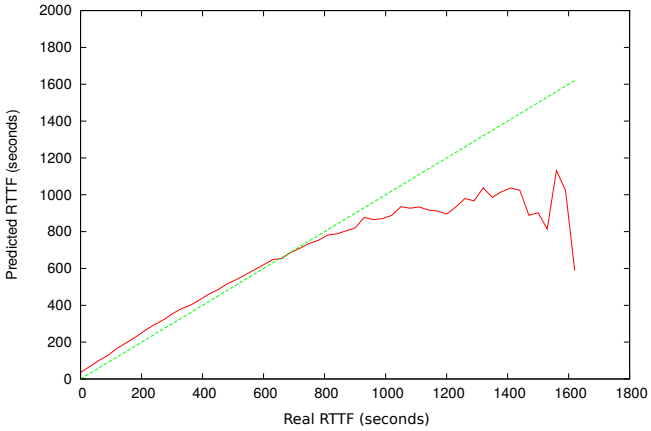

Fig. 2. Accuracy of the RTTF prediction model generated by F^{2}PM using M5P algorithm.

- A Managing Unit (MU), which manages the set of VMs, processes incoming messages from VMs and decides when to trigger the rejuvenation of a slave VM.

On all slave VMs, the following components are installed:

- A Communication Unit (CU), delegated to communicating with VMC;
- A Measurement Unit (MeU), which collects local measurements of the system features;
- A Local Managing Unit (LMU), which sends collected measurements to VMC and receives commands from VMC to start rejuvenating the (local) slave VM.

MU keeps a list of the couples of slave VMs. When the system starts up, the MU activates one slave VM for each couple of slave VMs, then it marks as *active* the activated slave VMs and as *stand-by* the other ones. The LBU forwards client requests only to the active slave VMs. The pool of VMs can be re-sized at run-time by adding or removing new couples of VMs.

In the PCAM architecture, as shown in Figure 1, dashed lines represent information exchanged among VMs and VMC. In particular, they enable VMC to implement the *On-line control loop*, by receiving values of monitored features by VMs, and sending to them the *rejuvenate* command. Solid lines represent requests coming from remote clients. These requests pass through the LBU, which forwards them to active the VMs.

B. Machine Learning-based RTTF Prediction

As we pointed out, the PU of VMC provides the predicted RTTF of a slave VM as a response to a query executed by the MU. The PU leverages on ML-based models, which are generated by using our previous result in [4], namely F^{2}PM, to predict the RTTF. F^{2}PM builds predictions model by exploiting a dataset of system feature measurements collected while monitoring VMs running in the presence of anomalies. For the sake of clearness, we provide a brief description of F^{2}PM. We nevertheless refer the reader to [4] for a comprehensive discussion on F^{2}PM.

F^{2}PM is based, as well, on a client-server architecture, namely on the *Feature Monitor Client* (FMC) and the *Feature Monitor Server* (FMS). FMS is installed on the VM that acts as a server, and it continuously collects system feature measurements of slave VMs. Further, upon a VM meets the failure condition, a failure event is registered and the VM is restarted. All these data are collected into a database. Then, in order to reduce the training time and the amount of data to be sent from LMUs of VMs to MU, system features are filtered using Lasso Regularization [8], so that only a subset of system features having stronger impact on the RTTF prediction of VMs are selected. The set of selected features is then enhanced with additional information (such as system feature slopes and the time to the next failure event) to capture more accurately the evolution of resource usage on slave VMs and the relation with the RTTF of VMs. Finally, the obtained dataset is fed into WEKA [9], in order to generate RTTF prediction models based on several ML techniques, including on Linear Regression [10], M5P [11], REP-Tree [12], Lasso as a Predictor [8], Support-Vector Machine (SVM) [13], and Least-Square Support-Vector Machine [14]. As an example, in Figure 2, we report a plot showing the accuracy of a RTTF prediction model generated by F^{2}PM using M5P algorithm. The figure shows a comparison between the real RTTF (ground truth, green line) with the predicted RTTF (red line). On the x-axis, we report the *real* RTTF, while on the y-axis we report the *predicted* RTTF. By the plot, we can see that, while the system is approaching the failure point (at time 0), the prediction accuracy of the model increases.

After generating the RTTF prediction models, F^{2}PM provides a number of indicators for each model, also including the mean and the relative absolute prediction error. In PCAM, we use these indicators to allow the user to select the most accurate model to be exploited for RTTF estimation. The selected model is therefore fed into PU of PCAM to be used at run-time.

F^{2}PM allows the user to customize the set of features that should be monitored. In our experimental study, we have selected the following ones:

n_{th} is the number of active threads in the system,
M_{used} is the amount of memory used by applications,
M_{free} is the amount of free memory in the system,
M_{shared} is the amount of used memory in buffers shared by applications,
M_{buff} is the amount of memory used by the underlying operating system to buffer data,
M_{cached} is the amount of memory used for caching data,
SW_{used} is the amount of used swap space,
SW_{free} is the amount of free swap space,
CPU_{user} is the percentage of CPU time spent by normal

processes executing in user mode,
CPU_{ni} is the percentage of CPU time spent by high priority processes executing in user mode,
CPU_{sys} is the percentage of CPU time spent by processes executing in kernel mode,
CPU_{iow} is the percentage of CPU time spent by processes waiting for I/O to complete,
CPU_{st} is the percentage of CPU time spent by processes waiting for services of other processes,
CPU_{id} is the percentage of CPU idle time.

Of course, system features used by PCAM are the same which are used by F^{2}PM to build the prediction models.

C. On-line Control Loop

Once the selected RTTF prediction model has been fed in PU, PCAM can preform the on-line control loop. Specifically, for each couple x of VMs, PCAM executes the following steps:

1) The LMU of the active slave VM, say $VM1_x$, collects local measurements of the set of system features and sends them to VMC.
2) Upon receiving measurements from the LMU of $VM1_x$, the MU of VMC queries the PU by using measurements as input, and retrieves the predicted RTTF of $VM1_x$. If the predicted RTTF is smaller than a (tunable) threshold T, then the MU performs the following actions:
   a) In the list of VMs, it marks $VM1_x$ as *stand-by* and the other slave VM of the same couple, i.e. $VM2_x$, as *active*;
   b) It communicates to the LBU the new list of active slave VMs;
   c) It sends to $VM1_x$ the *rejuvenate* command in order to start the rejuvenation procedure;
   d) It starts to receive measurements from $VM2_x$.

The diagram in Figure 3 provides an example of control flow to illustrate the on-line control loop control of PCAM in the case of one couple of VMs.

When a slave VM receives the *rejuvenate* command, it completes pending requests and starts the rejuvenation action. We remark that the MU switches the active VM of a couple before to send the *rejuvenate* command, thus new incoming request are immediately forwarded to the other VM of the couple. We further note that, when the active VM of a couple changes, it may be required, depending on the application, to execute some specific procedures before forwarding requests to the other VM, such as migrating data sessions, flushing in-memory data to shared databases, etc. Solutions for addressing these issues without affecting system availability as perceived by users have been discussed in [7]. We do not further deal with these issues, because they are orthogonal to the scope of this paper.

The threshold T used by MU can be specified by the user. Essentially, T defines a *safety value*, which is used by PCAM to determine the time instant when a VM has to be rejuvenated before the predicted failure time. On the one hand, this safety value allows to ensure that the VM can still execute pending requests before the actual failure time. Consequently, T has to be set at least in the same magnitude order of the request response time. On the other hand, we note that the value of T can be used to reduce the effect of MTTF prediction error on system availability. As an example, a low value of T entails that a VM is rejuvenated a few time before the predicted failure time. In such a scenario, even a very small overestimation of MTTF could prevent PCAM from rejuvenating a VM before the actual failure time. We note that, to cope with such a situation, some failure detection or SLA violation detection technique can be used to activate the other slave VM of the same couple. However, since these kinds of failures can be detected via standard failure detection and SLA violation techniques, we do not focus on this specific problem in our presentation. Indeed, the main goal of PCAM is to prevent the system from the occurrence of these kinds of events. As for a too high value of T, it might force too early rejuvenation a VM, even though it would still run with an acceptable performance for a long time. In Section IV, we discus in more details the effect of T leveraging results of an experimental study.

IV. Experimental Evaluation

In this section, we present an experimental study we carried out to the evaluate of the effectiveness of PCAM. We first describe the experimental environment used in our tests. Then, in order to select the machine ML to be used in our study, we briefly describe the results achieved in [4], where different ML models have been trained and evaluated using the methodology described in Section III-B. Finally, we show and discuss results of experiments carried out in the case of two different system configurations in terms of number virtual machines and combination of injection of anomalies.

A. Experimental Setting

We built our experimental environment in Amazon EC2, in Frankfurt region. In particular, we use `m3.large` instances, which offer 2 virtual CPUs, and 7.5 GB of RAM. Ubuntu Server 12.04 LTS is used as the virtualized operating system.

In our experimental study we use the well-know TPC-W Benchmark [15]. TPC-W is a transactional web e-Commerce benchmark. The system workload is generated by a number of users, which interact with the web shop through web browsers for searching, browsing and ordering books. TPC-W, in addition to mimic user behavior through a web page navigation graph, also defines the structure of all web pages, including images, and the database schema. The system workload can be configured by changing the number of concurrent users, and other parameters, such as the transaction execution mix or the database size. A comprehensive description of the benchmark can be found in the TPC-W specification document [16].

As discussed in Section III-A, PCAM includes a VM acting as a controller (VMC), and a number of slave VMs. In our experimental setting, slave VMs are equipped with Java SE Runtime Environment version 1.6. Each slave VM runs a Java implementation of the TPC-W benchmark [17], hosted by Apache Tomcat version 6.0. As for the database server, we use MySql version 5.1. The workload is generated via emulated web browsers. An emulated web browser simulates the presence of a user accessing web pages through a web browser. We run emulated browsers on an external 32-cores server in Munich, connected to the Amazon instances through the Internet.

We implemented the MU of VMC, the LMU of slave VMs and the CU in C language. The LMU relies on Linux `proc` filesystem to collect system features. CUs use TPC/IP sockets. As for the LBU, we used the open source load balancing tool Crossroads [18].

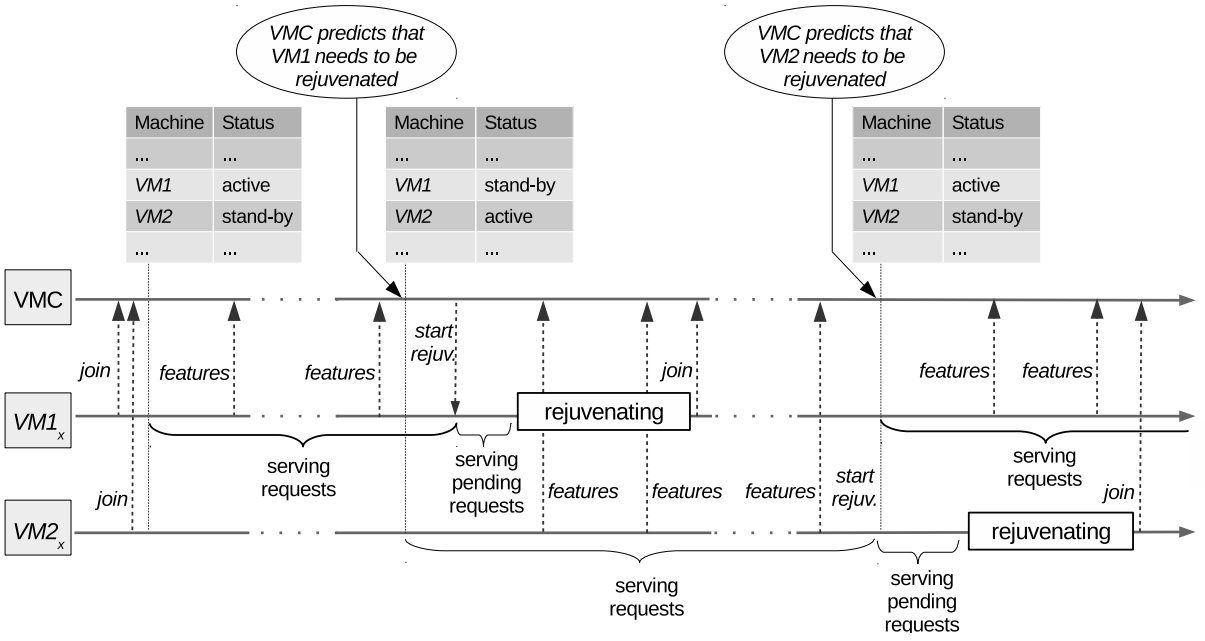

Fig. 3. MLSR-Framework control flow diagram with a couple of VMs.

We injected two kind of anomalies in slave VMs, i.e. memory leaks and unterminated threads. To this aim, we modified the Java implementation of the TPC-W. Specifically, when a user request was received by a slave VM, a new (dummy) Java object and/or a thread entering an infinite loop was generated with a given probability. In order to evaluate the ability of the framework to cope with scenarios with variable anomalies' injection rates, probabilities of generating a new object/unterminated thread in each VM, as well the size of generated objects, were randomly changed every time the VM was restarted after a (predicted) failure. This led to en execution scenario where VMs have been showing different anomaly occurrence patterns.

In our experiments, the failure condition of a VM was true if at least one of the following conditions was true: 1) both free memory amount and free swap space amount were less then 5% of the total memory and total swap space, respectively, 2) the average response time was higher than 4.5 seconds along 10 consecutive measurements by emulated web browser, and 3) the VM had not been sending measurements to FMS of F^{2}PM for more than 1 minute (which might suggest that the VM has crashed). We evaluated three supervised learning models for implementing MLP, i.e. Linear Regression, M5P (decision tree with the possibility of linear regression functions at the leaves), and the regression model achieved through Lasso regularization as a predictor. By our results, M5P has been proven as the most effective ML algorithm (both in terms of training time and prediction accuracy).

B. Framework Evaluation Results

We present experimental results for the cases of two scenarios. In the first one, we used only one couple of slave VMs, say c_1. In this scenario, client requests are processed by one active VM. Both memory leaks and unterminated threads are injected in slave VMs. When the predicted RTTF of the active VM is smaller than the threshold T, the MU executes the procedure for switching the active machine from $VM1_1$ to $VM2_1$ or viceversa (as related to the flow diagram in Figure 3). In the second scenario, we used three couple of VMs, say c_1, c_2 and c_3. Thus, in this case, client requests are processed by three active VMs (one per couple of VMs). The active VM of a couple is switched independently of the other couples of VMs. In this scenario, we injected both memory leaks and unterminated threads in $VM1_1$ and $VM2_1$. Conversely, we injected only unterminated threads (memory leaks) in $VM1_2$ and $VM2_2$ ($VM1_3$ and $VM2_3$). In both experiments, we used the prediction model generated by M5P algorithm.

Initially, in both scenarios, we set the threshold T equal to 300 seconds. Upon restarting a VM, the probability of generating anomalies for the VM is randomly selected in the interval $(0, 1]$, and the size of objects is randomly selected between 10 Kb and 1 Mb. As for the number of clients (emulated web browsers), we used 32 concurrent clients in the first scenario, and 64 in the second one. For both scenarios, we collected data related to all system features of the VMs, the response time measured by the clients and the predicted time to crash provided by the MLP.

We run the first experiment with 2 VMs for one week. In Figure 4, we show some results related to a time window extracted from the whole experiment for this first scenario. We report various measured features, namely number of active threads, free memory, used swap memory, and (total) CPU usage. Additionally, we report the response time measured by placing software probes in the Emulated Browsers, and the predicted RTTF for the VM that was activated upon each switching. By the plot, we can see that the accumulation of anomalies leads to a continuous decrease in free memory, with a subsequent increase in the usage of swap. Similarly, the number of active threads grows. The effects of these anomalies on the end users is shown by the response time, which grows as long as the effects of accumulated anomalies produce a performance degradation of the active VM. Further,

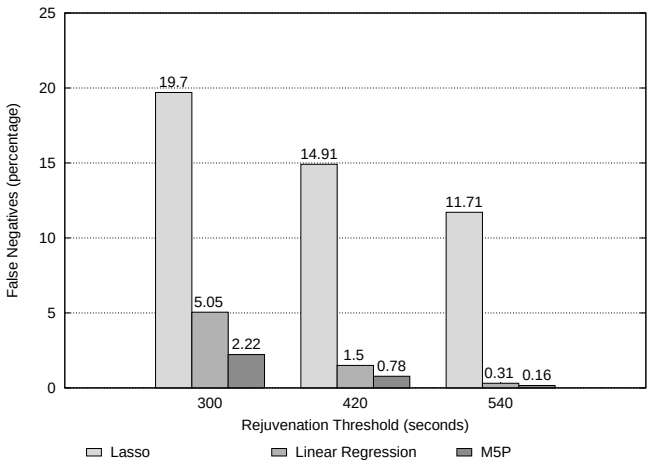

Fig. 6. Percentage of false negatives using different thresholds in the case of memory leaks and unterminated threads.

the predicted RTTF for the active VM shows a decreasing trend while increasing the amount of accumulated anomalies. Vertical red lines, in Figure 4, represent rejuvenation points, namely time instants where the active VM is switched. Indeed, after the occurrence of each vertical red line, we can see that the amount of available resources (e.g., memory free, threads, swap used) of the new active VM shows an anomaly-free state, and the predicted RTTF immediately increases. Yet, the response time measured by end users (emulated browsers) drops down. Particularly, we note that PCAM ensured an upper bound value of the average response time equal to 4.5 seconds (as we set in the VM failure condition).

In Figure 5, we plot measurements related to a time window of the experiment for the second scenario, where three couples of VMs are managed by VMC. As mentioned, the three couples of VMs are subject to different combinations of anomalies. Also in Figure 5, vertical dashed red lines represent rejuvenation points. By the results, we can see that the injection of different kinds of anomalies lead the different VMs to reach the failure point at different wall-clock time instants. Nevertheless, VMC is able to manage independently these different couples without any loss of timeliness in the rejuvenation action.

To complete the study, and to show the accuracy of PCAM, we measured the number of *false negatives*. As mentioned in Section III-B, F^{2}PM allows the user to define criteria to determine whether the system under monitoring should be considered as failed or not. This criteria are used, during the datapoints collection of the training phase, to mark specific datapoints as system failure points. At run-time, when VMC detects that the predicted RTTF is lower than the threshold T, a rejuvenation action of the active VM takes place. Nevertheless, due to prediction errors, it could be possible that the system fails although the predicted $RTTF$ is greater then T. This is exactly what we consider as a *false negative*. Namely, VMC treats the system as still working at an acceptable level, while it is actually not (i.e. it has already failed). In order to count the number of false negatives, we modified VMC in order to check if, based on values of features received by the active VM, the failure condition has been met.

In Figure 6, we report the percentage of false negatives we measured, for T equal to 300, 420 and 540 seconds, respectively, and related to prediction models generated using M5P, Lasso as a predictor and Linear Regression. Results show that the most-effective ML algorithm is M5P, providing a lower percentage of false negatives with respect to both Lasso and Linear Regression, confirming our previous prediction accuracy evaluation results in [4]. Nevertheless, in Figure 6, we can see that the higher the threshold T, the lower the number of false negatives. This is an expected result. In fact, as already mentioned in Section III-A, if T is set to a too low value, the effect of even small prediction errors might bring the system to the failure point, preventing VMC from performing a rejuvenation action before the system failure. Based on the results, we can see that with different values of T the percentage of false negatives significantly changes. This demonstrates that the value of T can be used for reducing the number of false negatives (possibly for eliminating them at all) and, as a consequence, for increasing the overall availability of the system. The counterpart of high values of T consists of increasing the overall system overhead due to the increase of the VM switching frequency and rejuvenation actions. However, in all our experimental scenarios, we observed that, also with higher value of T (i.e. 540 seconds) the increase of response time due to the higher VM switching frequency was negligible with respect to the overall response time reduction achieved with PCAM with respect the case when VMs are switched after a failure occurs.

C. Major Results

Results we discussed above show that PCAM can improve some system properties, including ones belonging to the category of *self-* properties*:

- *self-healing*: this property is guaranteed by timely forcing the system to an anomaly-free state, relying on the *self-rejuvenation* capabilities of PCAM;
- *self-optimizing*: this property is guaranteed by ensuring a response time of the system below an acceptable threshold, even in the case of accumulation of anomalies;
- *self-configuring*: this property is guaranteed by allowing an automatic reconfiguration of the system, also when couples of VMs are added/removed to/from the managed pool of virtual resources.

As shown, these three properties enable PCAM to support seamless execution of client-server applications deployed on a cloud infrastructure.

V. Conclusions and Future Work

In this paper, we present a ML-based framework to efficiently implement autonomic self-* properties and seamless execution of cloud applications in the case of client-server applications hosted in virtualized infrastructures. The framework uses prediction models that are generated using various ML algorithms, and can manage a cluster (with an arbitrary number) of virtual machines. PCAM works with applications that can be subject to different types of anomalies—as well as to different combinations of them—and applications that might be required to satisfy given SLA levels. By our results, we have shown that the direct effect on end users of PCAM is to ensure a response time below a given threshold, as well as to provide high availability.

In our future work we explicitly target the investigation of autonomic techniques to control (directly at run-time) the value

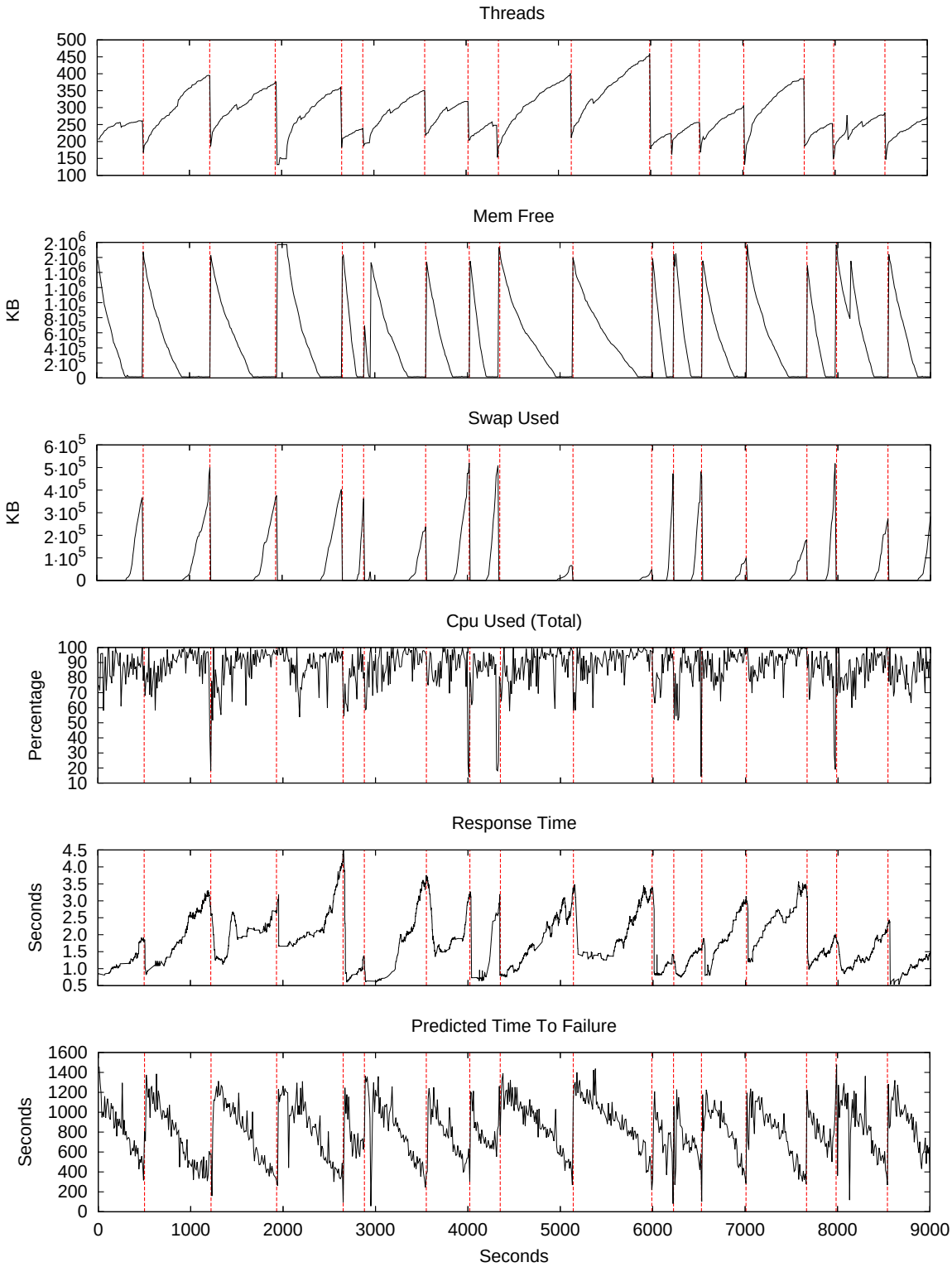

Fig. 4. System features, response time and predicted RTTF for the scenario with 2 VMs and Lasso (with reduced parameters) as a predictor.

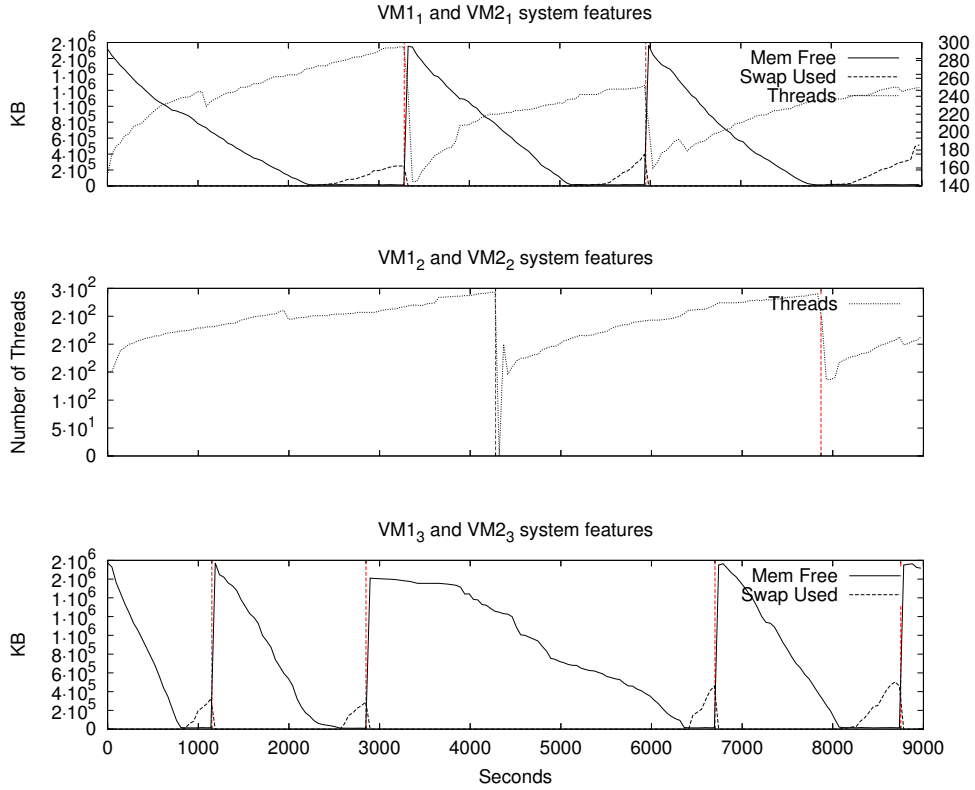

Fig. 5. System Features for the scenario with 6 VMs and Lasso (with reduced parameters) as a predictor.

of the threshold T, which plays a fundamental role in the self-optimization ability of PCAM. The goal will be to address various execution scenarios so as to significantly reduce the percentage of false negatives in order to meet SLA levels in terms of both availability and response time.

ACKNOWLEDGEMENTS

The research presented in this paper has been supported by the European Union via the EC funded project PANACEA, contract number FP7 610764.

REFERENCES

[1] S. Pertet and P. Narasimhan, "Causes of failure in web applications," Carnegie Mellon University, Tech. Rep. CMU-PDL-05-109, Dec. 2005.

[2] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software rejuvenation: analysis, module and applications," in *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, June 1995, pp. 381–390.

[3] D. Simeonov and D. Avresky, "Proactive software rejuvenation based on machine learning techniques," in *Cloud Computing*, ser. Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering. Springer Berlin Heidelberg, 2010, vol. 34, pp. 186–200.

[4] A. Pellegrini, P. Di Sanzo, and D. R. Avresky, "A machine learning-based framework for building application failure prediction models," in *Proceedings of the 20th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems*, ser. DPDNS. IEEE Computer Society, 2015.

[5] W. Dawoud, I. Takouna, and C. Meinel, "Infrastructure as a service security: Challenges and solutions," in *Proceedings of the 7th International Conference on Informatics and Systems (INFOS)*. IEEE Computer Society, March 2010, pp. 1–8.

[6] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam, "Critical event prediction for proactive management in large-scale computer clusters," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD. ACM, 2003, pp. 426–435.

[7] L. M. Silva, J. Alonso, and J. Torres, "Using virtualization to improve software rejuvenation," *IEEE Trans. Comput.*, vol. 58, no. 11, pp. 1525–1538, Nov. 2009. [Online]. Available: http://dx.doi.org/10.1109/TC.2009.119

[8] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society, Series B*, vol. 58, pp. 267–288, 1994.

[9] I. H. Witten, E. Frank, L. E. Trigg, M. A. Hall, G. Holmes, and S. J. Cunningham, "Weka: Practical machine learning tools and techniques with java implementations," 1999.

[10] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[11] Y. Wang and I. H. Witten, "Inducing model trees for continuous classes," in *Procedings of the 9th European Conference on Machine Learning*, 1997, pp. 128–137.

[12] H. A. Chipman, E. I. George, and R. E. Mcculloch, "Extracting representative tree models from a forest," in *IPT Group, IT Division, CERN*, 1998, pp. 363–377.

[13] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[14] J. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural Processing Letters*, vol. 9, no. 3, pp. 293–300, 1999.

[15] W. D. Smith, "TPC-W: Benchmarking an ecommerce solution," 2000.

[16] Transaction Processing Performance Council, *TPC Benchmark^{TM} W, Standard Specification, Version 1.8*. Transaction Processing Perfomance Council, 2002.

[17] T. Bezenek, T. Cain, R. Dickson, T. Heil, M. Martin, C. McCurdy, R. Rajwar, E. Weglarz, C. Zilles, and M. Lipasti, "Characterizing a Java implementation of TPC-W," in *Proceedings of the Third Workshop On Computer Architecture Evaluation Using Commercial Workloads*, 2000.

[18] [Online]. Available: http://crossroads.e-tunity.com/