Grant Agreement No.: 610764
Instrument: Collaborative Project
Call Identifier: FP7-ICT-2013-10

PANACEA

Proactive Autonomic Management of Cloud Resources

Deliverable 3.2: Machine Learning Framework and Global Architecture for Proactive Management

Version: v.1.0

Work package	WP 3
Task	Task 3.2
Due date	31/12/2014
Submission date	31/12/2014
Deliverable lead	IRIANC
Version	1.0
Authors	Dimiter R. Avresky
Reviewers	Michel Diaz, Eduardo Huedo

Abstract	In this deliverable, we present the Utilization of Machine Learning Techniques for predicting cloud resource utilization and anomalies, based on Distributed Machine Learning. Different research topics are investigated: Autonomic elements need to learn about their environment quickly; Dealing with noisy, dynamic environments; Coping with large amount of tuneable parameters; Systems will have many interacting resources, with multiple learners. We focus on creating Machine Learning framework and global architecture for Proactive management of cloud resources
Keywords	Machine Learning, Global Architecture, Proactive Management, Overlay Networks, Self* properties, Self-rejuvenation

Document Revision History

Version	Date	Description of change	List of contributor(s)
V1.0	31.12.2014	First version of the Deliverable	Dimiter R. Avresky (IRIANC)

Project co-funded by the European Commission in the 7th Framework Programme (2007-2013)		
Nature of the deliverable:	**R**	
Dissemination Level		
PU	**Public**	**X**
PP	**Restricted to other programme participants (including the**	
RE	**Restricted to bodies determined by the PANACEA project**	
CO	**Confidential to PANACEA project and Commission Services**	

EXECUTIVE SUMMARY

In this Deliverable we present the Machine Learning Framework and the Global Architecture for Proactive Management. We extended the ML in order to predict the violation of the Threshold of the Response time Webservers (Cloud Application). ML Framework generates predictions models with all features (monitored parameters of the physical resources) or with a reduced set of parameters, based on Lasso Regularization technique. The users can make a choice, based on the operational requirements of their cloud applications. Two versions of the Local Controller are proposed monitoring the parameters in VMs. Two modes of operations of the Inter/Intra – Autonomic Cloud Managers are presented for the realization of a control loop in their operations and Proactive Management of cloud resources. ML Framework & Global Architecture for Proactive Management are defined and the basic modules are implemented. The ML Framework can be deployed on both Hybrid Clouds and Private Local Virtual Infrastructures. Intercommunication among different deploys is ensured by Inter-ACMs communication.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

AM	Autonomic Manager
ACM	Autonomic Cloud Manager
ML	Machine Learning
VM	Virtual Machine
RT	Response Time
M5P	Rational Reconstruction of M5
SVM	Support-Vector Machine
SVM2	Support-Vector Machine to the power of 2
REPTree	Replication Tree
RTTF	Remaining Time to Failure
RTTC	Remaining Time to Crash
RTTH	Remaining Time to Hit the Response Time Threshold

1 INTRODUCTION

We extended the ML in order to predict the violation of the Threshold of the Response time Webservers (Cloud Application). ML Framework generates Predictions models with all features (monitored parameters of the physical resources) or with a reduced set of parameters, based on Lasso Regularization technique. The users can make a choice, based on the operational requirements of their cloud applications. Two versions of the Local Controller are proposed monitoring the parameters in VMs. Two modes of operations of the Inter/Intra – Autonomic Cloud Managers are presented for the realization of a control loop in their operations and Proactive Management of cloud resources. ML Framework & Global Architecture for Proactive Management are defined and the basic modules are implemented. Intra-ACMs forming Private Clouds and communicating via Inter-ACMs are capable of creating Federated Clouds .

We will explicitly rely on Machine Learning (ML) algorithms [1], [2] which, by using data measured from the running web server instance(s), will be trained by relying on specific training sets obtained by a preliminary empirical experimentation phase. Given the large amount of variables which could affect a system' s functioning, we will specifically select the most significant ones, by relying on the Lasso approximation model [3], showing nevertheless a low value of Mean Absolute Error of the prediction.

We created a working prototype of a system that allows self* properties (healing, rejuvenation, reconfiguring) and seamless application execution to be achieved.

We note that this framework can be used for any application (not only web servers) that uses a lot of resources, requires a huge amount of uptime.

The building blocks of ML Framework & Global Architecture for Proactive Management are implemented and via the Overlay Networks scalable and resilient to partitioning Intra/Inter virtual networks can be accomplished.

In [4], the following set of requirements are used for selecting Web services in large Data centers in the world.

Requirements: Availability-The Probability that the Web Service is operational; Price; Popularity; Data-size –the size of the Web service invocation response; Success –probability that the a request is successfully completed at the server side and the corresponding response is successfully received by the service requestor**;** Response time – the average time duration between a service user sending a request and receiving a response; Overall Success probability – the average value of the invocation success probability of Web service observed by different service users; Overall Response time – the average value of the response-time of a Web service observed by different service users.

The Overall Success probability and Overall Response time provide relevant information for better Web service selection. The Overall Response time has a primary role for selecting the Web service and the conventional maximum upper bound is 3 sec.

In [5] The Cloud provider (CloudWeaver) optimizes your cloud environment and presents experimental results for upper bounds for the average Web servers response times that are acceptable for the users. i. e. in the interval between 2 and 4 seconds.

The majority of users are leaving these services, when these boundaries are not guaranteed.

Commonly, the anomalies (memory leaks and unterminated threads) in the VMS, performing the Web services, and the workloads are the reason for the problems.

The same approach can be applied for real-time applications with pre-determined execution boundaries.

On the other hand, Run Time to Crash (RTTC) is another important parameter to be predicted for performing the proactive management of cloud resources.

Based on these consent requirements, we will further develop the ML framework for a proactive management of cloud resources.

Machine Learning algorithms can be classified into two broad categories based on the nature of input and expected output of the algorithms [6] [7]: *supervised* [8] and *unsupervised* [9] *learning*.

In our work we are using supervised learning, which algorithms requiring well labelled dataset. For example, each instance in a training performance dataset is assumed to belong to one of several classes e.g. normal or abnormal performance behaviours.

ML framework must take into consideration specific cloud requirements, as described below, in order to implement the proactive management: Scale [10] [11] [12], Multi-tenancy [13], Complex Application Architecture [10] [11] [12], Dynamic Resource Management, [10] [11] [12] [13], Autonomic Management [10] [11] [12] [13]; (*Delayed detection* and *manual resolutions* will not feet cloud model as they will be the reason for financial penalties and performance failures. Consequently, ML Framework must *dynamically* and *proactively* manage the Cloud resources.)

2 ML-BASED PREDICTION FRAMEWORK

The Machine Learning Framework presented in Deliverable 3.1 [14] has been augmented with some additional steps, and its final incarnation is depicted in Figure 1. Training is required when hardware and/or software configurations change.

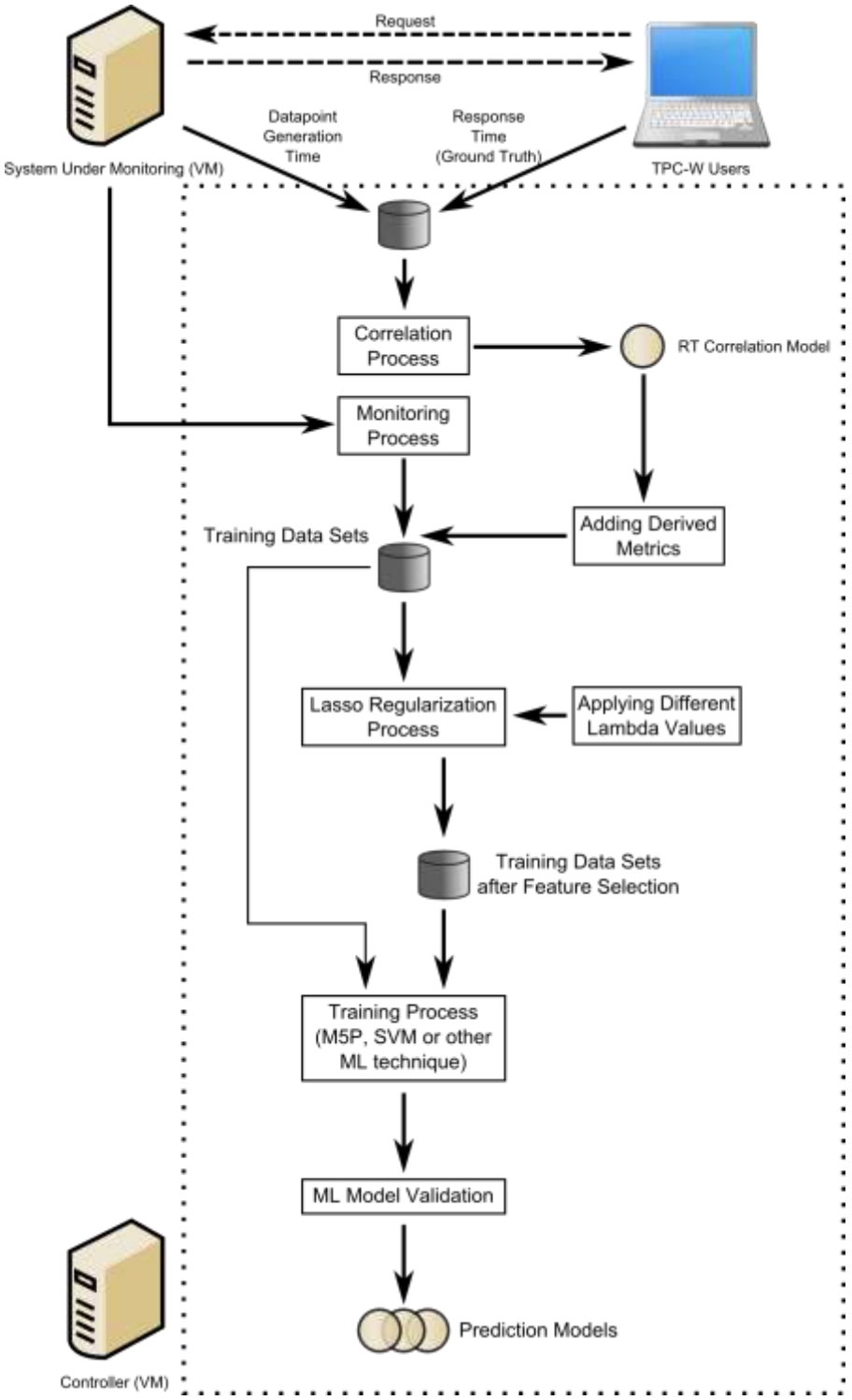

Figure 1: Machine Learning Framework

In this Section, we highlight several enhancements, which we have implemented with respect to the Framework description provided in Deliverable 3.1 [14], and we additionally provide a short recall on the functioning of the overall Framework, for the sake of clarity.

The ML Framework has been extended in order to explicitly take into account the Response Time, as seen by the end users, of any virtualized web application hosted on the machine under monitoring. This allows to catch the effects of unterminated threads more promptly. During the offline learning process, we rely on a set of clients to generate the workload on the server. In this specific scenario, since clients are actually controlled by the system administrator carrying on the training process, it is possible to measure the actual response time as experienced by the clients.

Given the fact that during the injection of anomalies we are relying on the collection of hardware features to build the initial training data set (see Deliverable 3.1 [14] for a thorough description of this process, and Section 2.2 for a recall), at this point, we are able to build an extended set of training data which are composed of the actual Response Time (the ground truth) and the difference between the generation timestamp of two consecutive data points (which we will refer throughout this Deliverable to as *Generation Time* or *GenTime*).

By using this data, we rely on WEKA [15] to carry on (using the fast Linear Regression) a *Correlation Process* to build a Response Time (RT) Correlation Model. In Figure 2, we report experimental results of this correlation process. Specifically, as we can see by the plots, both Generation Time of data points and Response Time (ground truth) are increasing when the system is suffering from leaks and unterminated threads.

By our results and experimentation we found out that WEKA is giving good results with respect to other tools available, like RapidMiner [16] and Massive Online Analytics (MOA) [17]. In the future, if at a certain point of the project WEKA fails to be successful, we will integrate these or other tools in our Framework. In fact, the modularity of the Framework allows to replace any of the (implementation-specific) steps with other ones, providing different implementations of the same ML algorithms.

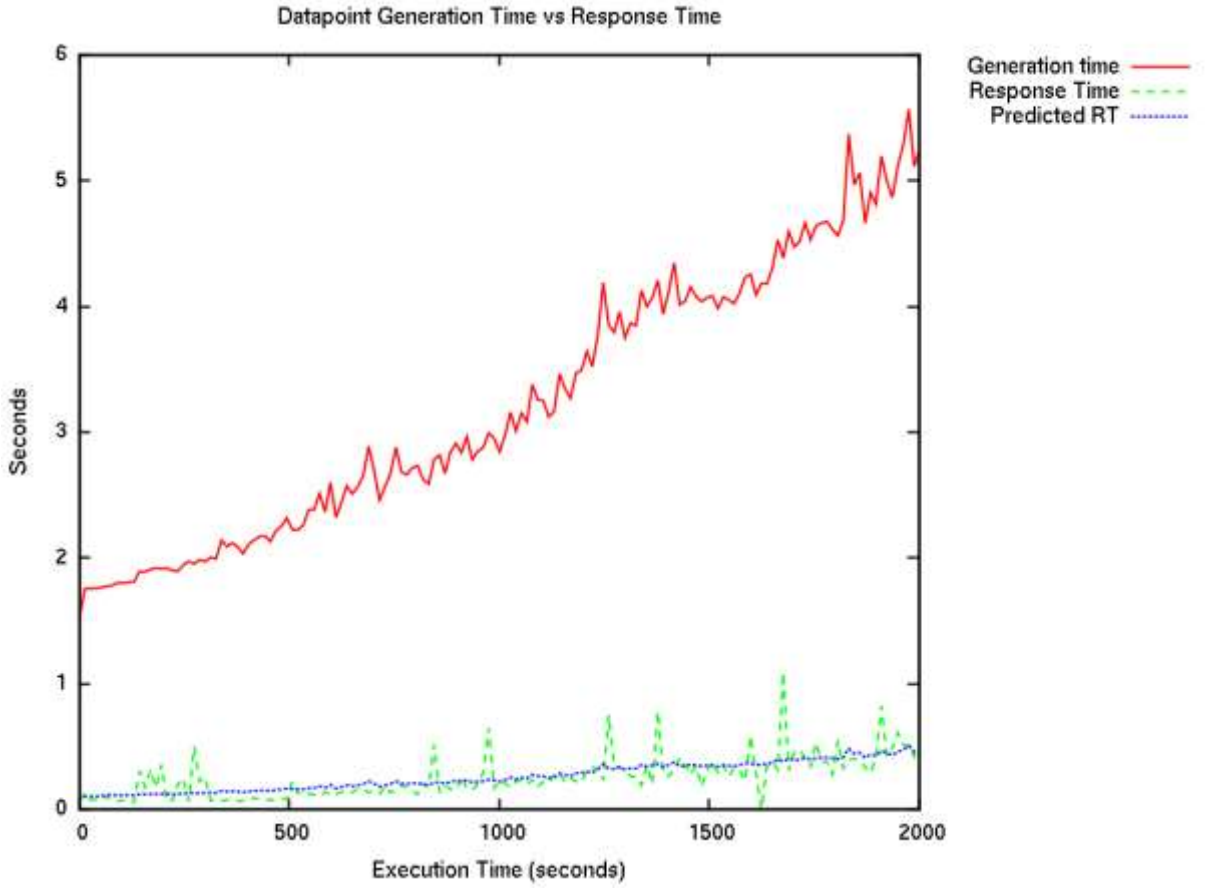

Figure 2: Correlation process to estimate Response Time

The correlation process is able to build a model for the *Predicted RT*, namely an estimation of the actual RT experienced from remote clients, starting only from the measurement of Generation Time.

Under the assumption that the system under monitoring will run the same application as the one used during this initial training phase, we can exploit (during the latter monitoring step) to use this correlation model in a twofold mode:

1. On the one hand, this correlation model can be used during training to detect whenever Remote Clients are experiencing a RT, which exceeds a given threshold. When the clients hit this threshold, we consider that the system is violating some SLA, and therefore, even though we have not yet experienced a real crash, we handle this situation exactly as if the system had already crashed. In fact, the system might still stay alive for a long time (although it will eventually crash), yet we are no longer able to provide a service with a reasonable quality. Therefore, we consider this analogous to a crash, and in this specific case, we compute the *Remaining Time to Hit the Threshold on Response Time* (RTTH). RTTH will be used during training phase in a way similar to the more traditional RTTC.
2. At this point, the Generation Time information will be used, during training, as an added derived metric. In fact, this is not a proper hardware parameter, yet it can be used to let the system predict when the SLA will be violated, thus allowing the ML Framework to learn at the same time the RTTC and RTTH, which will be treated evenly.

The second enhancement proposes a choice for the user to select how to build final prediction models. In fact, using Lasso regularization allows to reduce the number of parameters used to build prediction models using WEKA. This has a twofold effect: on the one hand, the time required to build the model is reduced, and on the other hand the impact of monitoring resources during the online prediction to determine whether a system should be rejuvenated or not is reduced. Yet, as it will be shown in Section 3, this reduction of the number of parameters can affect the final precision of the prediction model. Therefore, as depicted in Figure 1, the end user of the Framework is given the freedom to choose whether to build prediction models using all parameters or only the ones selected by Lasso.

As an additional note, using Lasso for online prediction, as it will be clearly shown in Section 3.8, will give predictions, which suffer from false positives. However, they are extremely conservative and never produce any false negatives. This has, as well, implication on systems which demand highly availability. In the end, the user can decide whether to use for prediction Lasso (targeting, e.g., environments requiring high availability), or models built using different learners using either the whole set of parameters (e.g., in case of the possibility to support longer training times) or only the ones selected by Lasso. In this sense, the ML framework can be fully tuned at startup, in order to detect which execution pattern shown in Figure 1 will be actually followed.

2.1 Recall on Feature Data Collection

As mentioned before and in Deliverable 3.1 [14], the initial step to build prediction models is to collect runtime data from the system under monitoring. We report in Figure 3 the abstract architecture of the system when collecting this data.

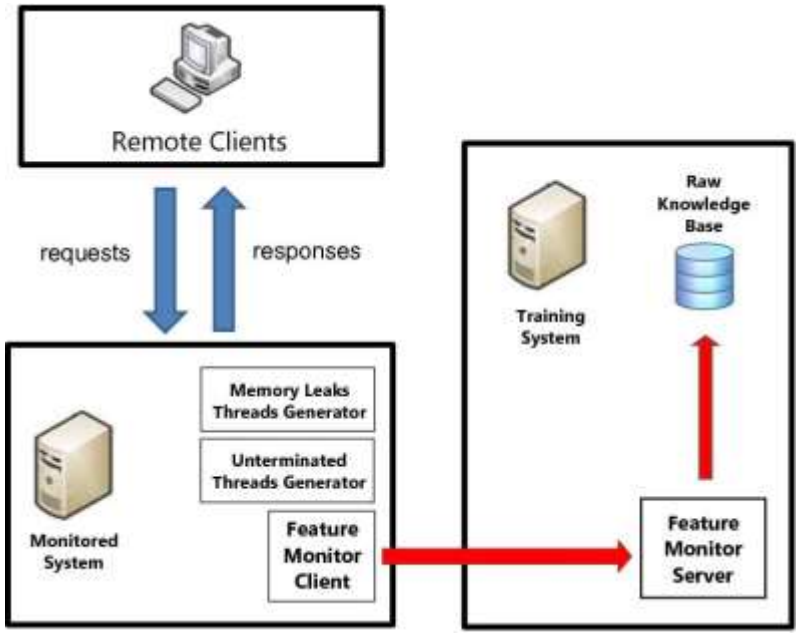

Figure 3: Abstract Architecture for Data Collection

To build the initial database file (Raw Knowledge Base), namely the input file to the ML Framework, two different anomalies can be injected in monitored system to mimic a software environment which eventually becomes faulty, namely *memory leaks* and *unterminated threads*. To correctly mimic the behaviour of a system affected by software errors, anomalies are injected according to statistical distributions. This is done by relying on two utilities, namely a Memory Leak Generator and an Unterminated Threads Generator, installed and run on the Monitored System.

Memory leaks are generated by allocating periodically a variable-size contiguous chunk of memory and writing dummy data into it. Writing data is essential to mimic a faulty implementation, as otherwise the underlying Linux kernel would not really allocate physical memory for the buffer, yet only virtual memory would be allocated, which on its turn would not occupy space.

This is because the Linux Kernel, internally, handles memory on a per-process basis, via the `struct vm_area_struct`. This structure defines a memory VM memory area. There is one of these per VM-area/task. A VM area is any part of the process virtual memory space that has a special rule for the page-fault handlers. To make the long story short, whenever new memory is requested, if the underlying `malloc` library has not enough space, it is requested to the kernel via a `mmap` call. The effect of this call is not to allocate memory, but only to *reserve* it. Therefore, the memory is actually allocated only upon the first write on it.

A faulty implementation of a system is expected to allocate memory for real usage, which is later not released. Then, to mimic this behaviour we must ensure that the memory is actually *allocated*, not only *reserved*. Therefore, our dummy write on it allows for an effective generation of memory leaks.

We exploit two statistical distribution to implement memory leaks. On the one hand, we rely on a uniform distribution, in the interval [1KiB, 10 MiB], to define what is the size of the current leak. This is because, in general, application require both small-size buffers and large-size buffers to carry on their work, and both these kinds of buffers could be not released by a faulty implementation. The second statistical distribution is an Exponential one, which is used to draw the time to wait before the next memory leak occurs. The mean of this exponential distribution is randomly drawn at each leak-generation step using again a uniform distribution in between [0.5, 10] seconds. This allows us to mimic the execution of the "faulty portion" of the software more or less often. In any case, relying on statistical distribution and repeating the experiment a very large number of times allows us to generate a pattern which can be significant for a specific implementation of a faulty software, which is exactly the goal of our use case.

Concerning memory leaks, the above discussion leads to an injection of memory leaks which is described by the cumulative function (referred to a single execution of the memory leak injection pattern on one of our 8GiB-memory virtual machine) which is reported, for the sake of clarity, in Figure 4, where on the x-axis are reported seconds.

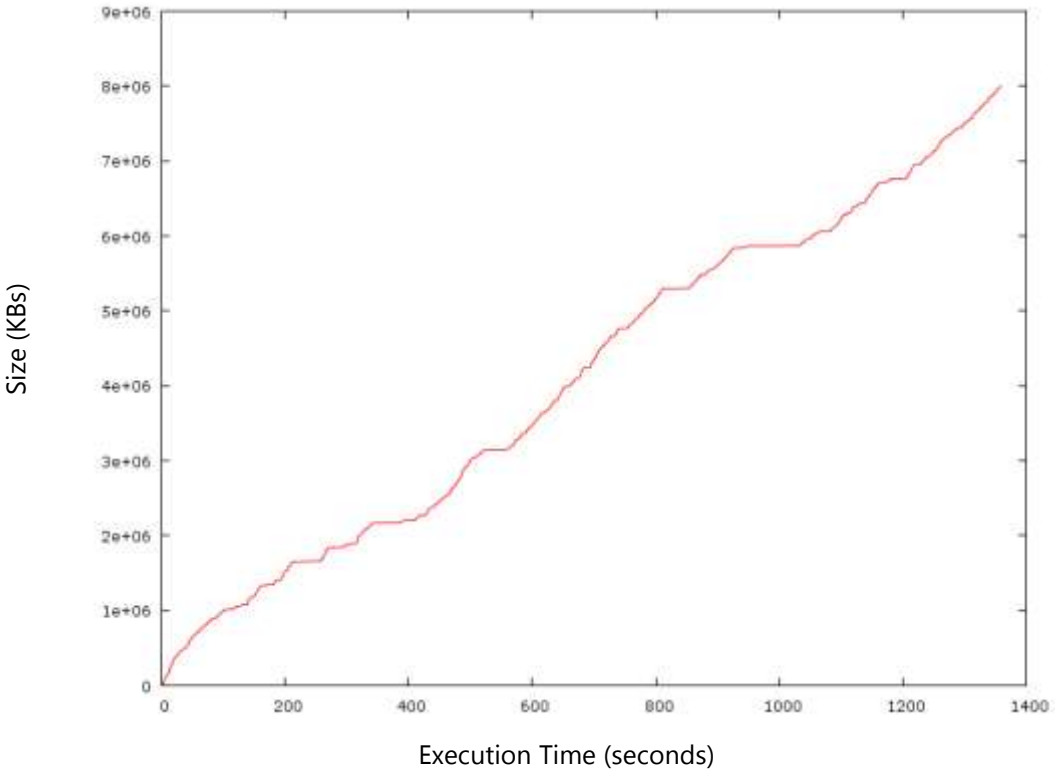

Figure 4: Memory leak injection

An *unterminated thread* is essentially a thread, which has done useful work for a certain amount of time and which, at the end of its life, fails to be terminated. This can be again related to faulty software implementations, and their effect can be disturbing for the correct execution of the system.

Similarly to the case of memory leaks, we have resorted to one Exponential statistical distribution to draw the time spanning between two consecutive generations of unterminated threads. The average of the exponential distribution is drawn uniformly at random at each injection step from the interval [50, 200] seconds. This higher value is related to the fact that an unterminated thread actually has performed useful work at the beginning, and therefore, the faulty code not releasing it can be executed less often, actually after having carried out the useful work.

2.2 Recall on the ML Framework Steps

The remainder of the ML Framework acts exactly as previously described in Deliverable 3.1 [14], as the integrations have been explicitly developed in order to be fully compliant with the previous version of the Framework, thus allowing retrocompatibility with models built using any version of the Framework. For the sake of clarity, in this Section we recall the workflow of the Framework, to shown how models are built after that the client/software for feature monitoring has been used to build the initial database file. Compared to the input features described in Deliverable 3.1 [14], we have augmented its set by adding the number of active threads on the system and the Generation Time of datapoints, to allow for a more accurate

learning.

The Machine Learning Framework is based on a set of utilities, which mostly just require a standard C compiler on a POSIX system, with `pthreads`. Any Linux distribution will likely meet these dependencies. Nevertheless, some steps of the learning Framework will require an additional set of dependencies, namely:

- gnuplot [18] (to generate plots of the prediction models)
- WEKA [2] (at least version 3.7, to run some of the learning algorithms)
- Matlab [19] (to run data regularization, and to build the Lasso prediction model)
- LaTeX [20] (for automatic generation of training reports)

Failing to satisfy any of these dependencies will prevent some steps to be correctly carried out. Some of these steps can be nevertheless disabled during the configuration of the framework. The corresponding steps can be activated as well depending on the dependencies which are met on a specific machine.

Using the Framework is quite straightforward. To compile the tools, simply issue `make` in the main folder of the package. Then, since all the ML Framework steps are governed by the Bash `FRAMEWORK.sh` shell script, several variables should be set in it, according to the following meaning.

- `database_file`: this is the database file generated by the feature collection architecture keeping the data obtained from the machines under monitoring when injecting anomalies and making them crash. By default, the output name of the file is db.txt, but here a different name can be specified for convenience.
- `lambdas`: this is the set of lambdas used to build prediction models with Lasso. Defaults range from very small values to very large ones, allowing to extensively test the behaviour of Lasso
- `WEKA_PATH`: this must be set to the path where WEKA is installed on the system. If this variable is not correctly set, the steps `run_weka_*` should be later disabled.
- `lasso_algorithm`: this specifies which Lasso variant should be used when building models using Lasso. Supported variants are `grafting`, `iteratedRidge`, `nonNegativeSquared`, and `shooting`. The default value for this parameter is `iteratedRidge`, as by our experiments it has been showing the best and most stable results.
- `build_with_lasso_parameters`: if this configuration variable is set to `true`, then WEKA uses only parameters selected by Lasso for final model building. On the other hand, if it is set to `false`, all the parameters are used.

Then, the steps to be carried out by the Framework can be specified with a set of additional variables. Setting a variable to `false` disables the step, while setting it to `true` enables it. The available steps are:

- `run_datapoint_aggregation`: this step generates aggregated datapoints from the initial database file, which can be later used for training. The Framework relies on aggregated datapoints, so this step is mandatory, and could be disabled only if it has been run at least once.

The default aggregation time interval is 15 seconds. The aggregation is done according to the scheme shown in Figure 5.

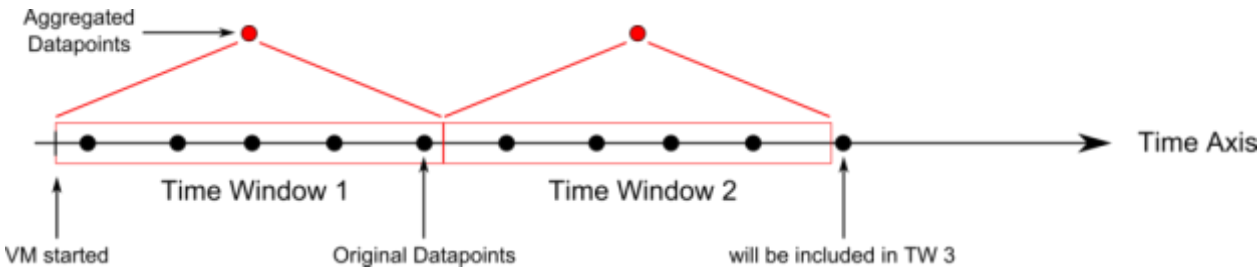

Figure 5: Datapoint Aggregation Scheme

Essentially, each input datapoint (shown in black in the Figure) is generated at a certain time point during the lifetime of the VM, and keeps some information about the current state of the system. The Generation Time information is used to virtually place the datapoints on the original generation time axis. Then, a time window of size equal to the threshold is placed starting at $T_0 = 0$. All the original datapoints falling in this time window are used to generate an aggregated datapoint, meaning that all the values of the original datapoints are averaged in the aggregated datapoint.

An additional goal of this step is to give an additional set of derived metrics to be used as features for model building, namely slopes and Generation Time. Then, for each of the original measurements, slopes are computed according to the following formula:

$$slope = \frac{start - end}{n} \quad (1)$$

where *start* and *end* are the values (for each measurement) of the first and last original datapoint falling in the current time window

- `run_lasso`: this step runs the selected Lasso algorithm for all the specified lambda values. Lasso is implemented as a set of Matlab scripts, which perform *lasso regularization*. For each element λ of vector *lambdas*, this steps computes the value of the vector β, whose elements are the *weights of the* vector x_j which minimizes the following objective function:

$$\frac{1}{n}\sum_{j=1}^{n} V\left(y_j, \langle \beta, x_j \rangle\right) + \lambda \|\beta\|_1 \quad (2)$$

where n is the number of data points included in the output of the script *aggregate.py*, x_j is a vector of values of input features (independent variables) of each data point, y_j is the associated value of the dependent variable (remaining time to failure of the system) for the specific data point, and $V(y_j, \langle \beta, x_j \rangle)$ is equal to $(y_j - \beta^T x_j)^2$.
For each value of λ (which is specified in the script in the `lambdas` configuration variable), the calculated vector β includes a (sub-)set of non-zero elements. Only features to which corresponds a non-zero element of the vector β are subsequently used as input features to WEKA for building machine learning models, in case the user has set the variable `build_with_lasso_parameters` to `true`. Generally, while increasing the value of λ, lower values of elements of the vector β are selected. Thus the effect of using higher values of λ is, generally, the reduction of the selected features to be used in the machine learning models.

- `apply_lasso`: this step builds derived datasets according to the weights computed by Lasso for each lambda value. These datasets are needed to build prediction models in case the user wants to use only parameters selected by Lasso.
- `plot_original_parameters`: this is a convenience step to generate plots of the trend of original parameters (see Figure 8 for a sample output)
- `plot_lasso_as_predictor`: this is a convenience step to plot prediction models built using Lasso (using different values of λ)
- `run_weka_linear`: this step builds the Linear Regression model using WEKA and plots the prediction model built.
- `run_weka_m5p`: this step builds the M5P model using WEKA and plots the prediction model built.
- `run_weka_REPtree`: this step builds the REP Tree model using WEKA and plots the prediction model built.
- `run_weka_svm`: this step builds the SVM model using WEKA and plots the prediction model built.
- `run_weka_svm2`: this step builds the SVM2 model using WEKA and plots the prediction model built.
- `evaluate_error`: this step computes the error of all prediction models (see Table 2, Table 3, Table 4, Table 5, and Table 6 for sample output)
- `generate_report`: this final steps builds the report with all the information obtained. This requires LaTeX and to have run all the previous steps.

2.3 Coordinator and Load Balancer for distributed VMs

Training Data sets from multiple (distributed) VMs that are under different anomalies are shown in Figure 6. It is based on the abstract architecture of the system, as shown in Figure 3, when collecting this data.

There are two cases for set up the VMs:

a) Installing all VMs in HPs server, controlled by the VMware hypervisor (Local Virtualized Infrastructure).

b) Mounting a portion of VMs in HP Server and controlled by the VMware hypervisor. The rest of VMs are placed in the Cloud (Hybrid Cloud).

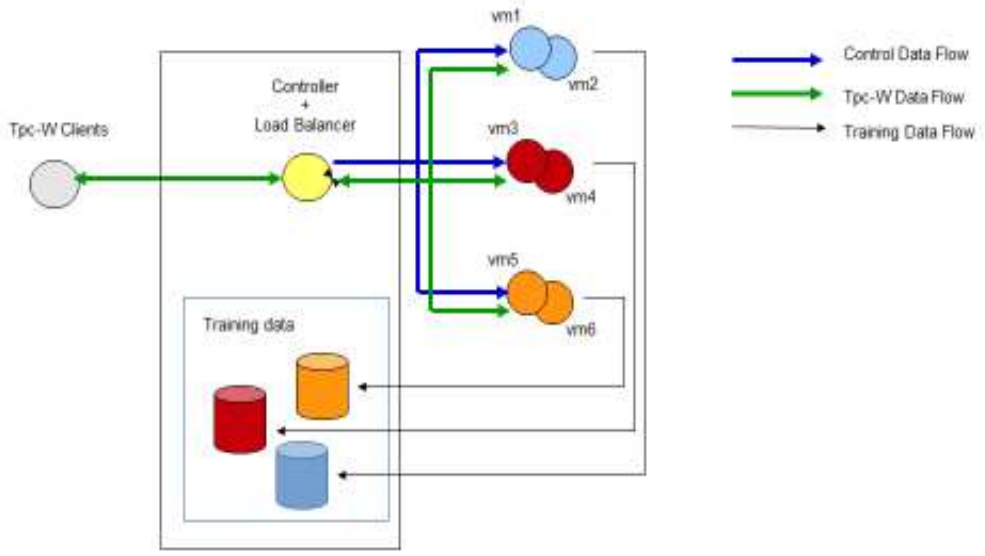

Figure 6: Controller and Load Balancer for VMs

2.4 Dynamic Reconfiguration Flow Diagram

The models built using either Lasso or WEKA are automatically embedded (using some C header files) in the controller. Specifically, the user can setup the controller in order to use any of the generated models to carry on the online monitoring process targeted at prompt proactive rejuvenation.

Figure 7 shows the behaviour of the system when running under the supervision of the controller (VM1). This is actually the default initial behaviour, in which the system starts running. When VM1 is activated, it sets up its Feature Monitor Server (FMS) module and Proactive Control Module Server (PCMS). When VM2 and VM3 are activated, they set up their own Feature Monitor Client (FMC) and the Proactive Control Module Client (PCMC).

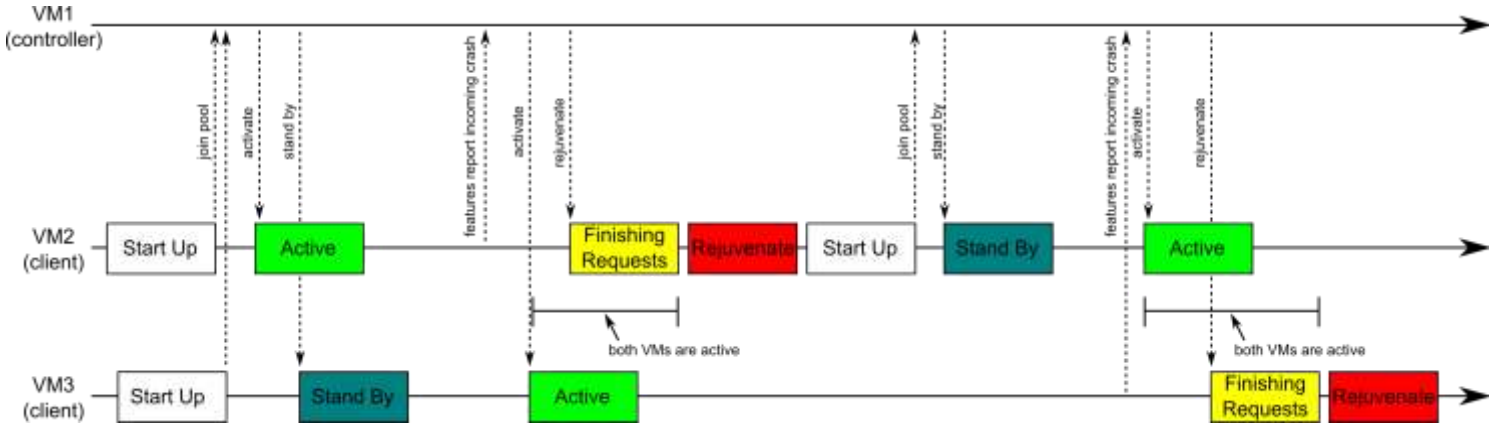

Figure 7: Dynamic Reconfiguration Flow Diagram (Soft Rejuvenation)

In particular, VM2 and VM3 enter immediately the Start Up state, which tells the PCMC to emit a *join pool* control message towards the PCMS. At this time, VM1 processes this control message by building a (dynamically changing over time) set of VMs, among which one is selected as the current active one. VM2 receives the *activate* control message (which brings VM2 into the Active state), while VM3 receives the *stand by* control message, (which brings VM3 into the Stand By state). We emphasize that this same protocol can scale to any number of VMs used to build private clouds.

The packet forwarded on VM1 is able to exploit the information stored by PCMS on the same VM to know what is the currently active VM. In this way, users start sending requests, which are forwarded to the current active VM.

FMCs on both VM2 and VM3 then start sending performance measurements to the PCMS. When PCMS detects that the currently active VM is about to fail[1], it sends an *activate* control message to VM3, and a *rejuvenate* control message to VM2, and updates its internal state of the VM pool. Then, the packet forwarder detects this change, and it forwards new requests to VM3. In the meantime, VM2 finishes serving pending requests. To this end, VM2 waits for these requests to be served, and only after this transition phase it rejuvenates itself. After the rejuvenation, VM2 reaches the Start Up phase, connects to PCMS on VM1 and it is added to the pool of available VMs.

[1] This prediction is done by using the prediction model generated by ML Framework. The term *about* should be read as *the RTTC/RTTH is lower than a specified threshold*. We have chosen a threshold which is able to take into account the time required by the spare VM to correctly switch to the Active state.

3 MACHINE LEARNING FRAMEWORK: THE USED PREDICTION MODELS

To complete the experimental results of the Machine Learning Framework presented in Deliverable 3.1 [14], we have run as well experiments with the TPC-W server when injecting both memory leaks and threads, using a variable number of threads (in between 8 and 64). This allows us to evaluate the Framework under a more varied workload, and to assess the validity of the approach even when more fluctuations are present in the system.

In the final experimentation, as it will be described in Section 6, we have relied on both the models already shown in [14], and on this new model, to enforce proactive rejuvenation.

We have collected data related to the behaviour of the TPC-W server up to the collection of **25 MiB of raw database points**. This data has been feed into the Machine Learning Framework to generated the aggregated database file as described before.

Figure 8 shows the average system parameters (before feeding them into Lasso), as measured during the collection. The dashed curves (CPU usage and inter-datapoint generation time) are to be read against the right-hand side y axis, while the solid ones (memory usage) and the dotted one (number of active threads) are to be read against the left-hand size y axis.

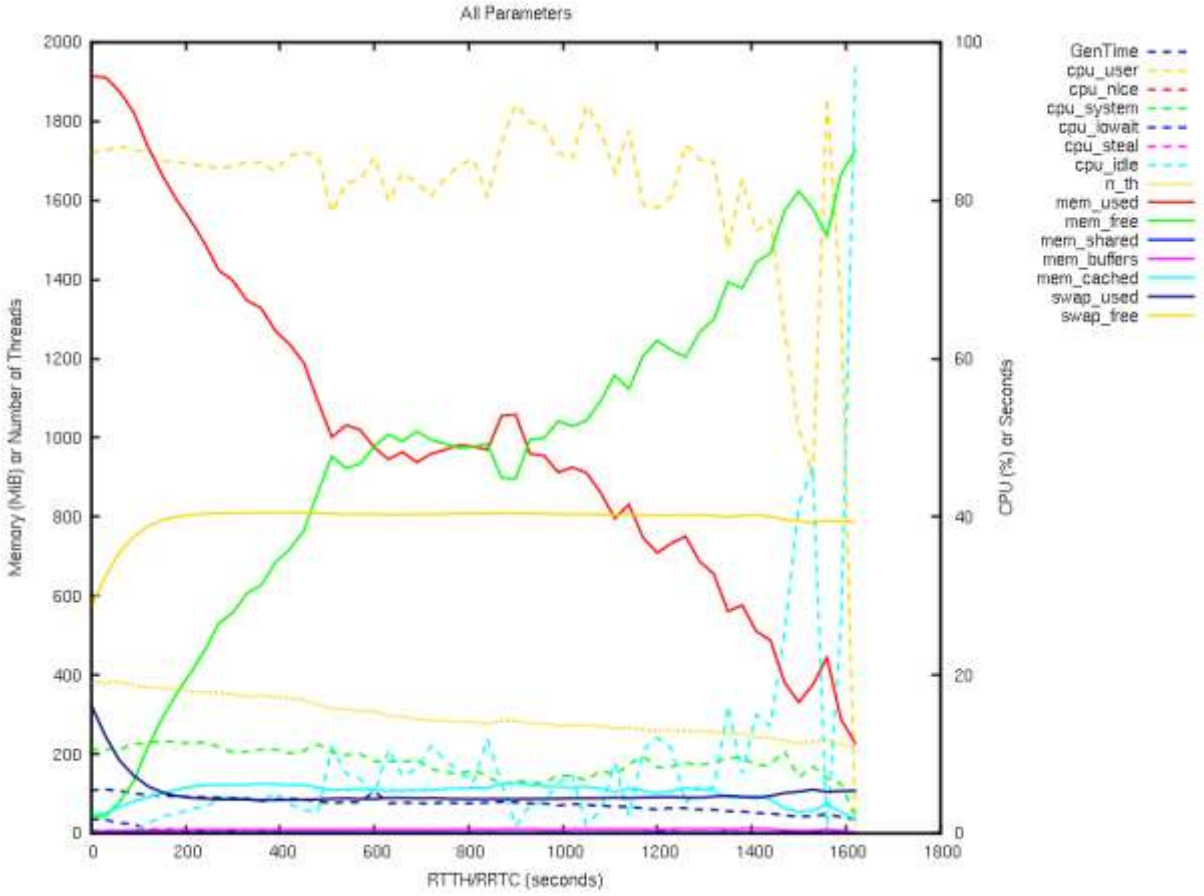

Figure 8: System parameters (memory leaks)

On the x-axis, we report the RTTC and/or the RTTH, depending on whether the crash of the system was due to memory exhaustion or to a too high inter-datapoint generation time (we recall, by the discussion in Section 2, that the inter-datapoint generation time has been correlated to the response time as seen by TPC-W clients, in order to estimate the moment where a specific SLA level was violated—which we have been calling "threshold hit").

This data shows that under this variable workload, the system can become very stressed, even after a short period of time.

In Figure 9, we report the Response Time of the TPC-W web application when injecting memory leaks and unterminated threads in the system.

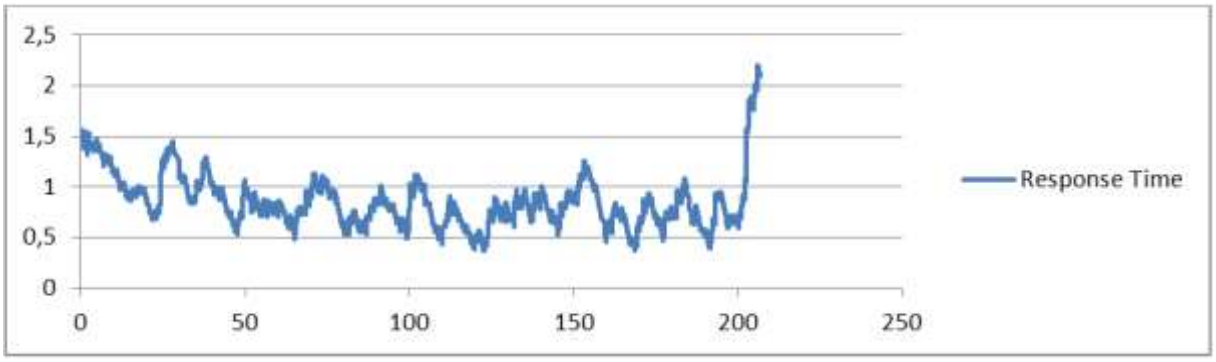

Figure 9: TPC-W Response time

3.1 Parameters Selected by Lasso

The first step of the ML Framework is to apply Lasso Regularization in order to try reducing the number of parameters used for building the model, while keeping a sufficient accuracy. In Figure 10 we show a graphical representation of this selection, showing how many parameters where selected when increasing the value of λ.

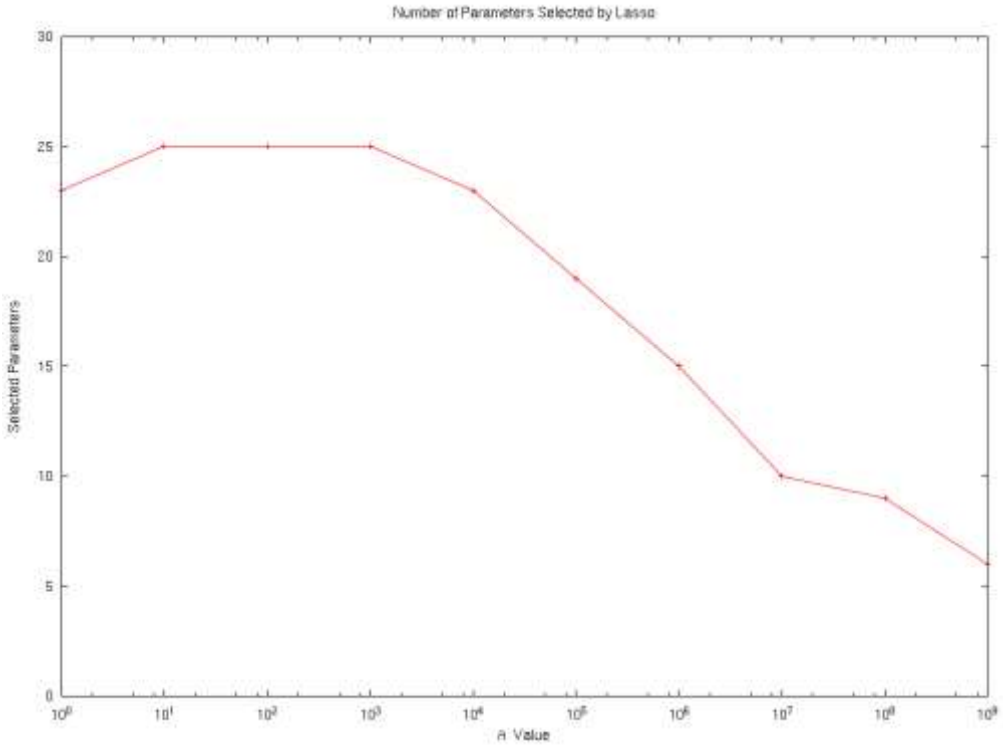

Figure 10: Number of Parameters selected by Lasso

According to the results, we can see that Lasso is actually able to reduce the amount of parameters to be used for subsequent model building, ranging from an initial value of 22/23 parameters, to a final value of 6. As we mentioned, this allows to reduce the time for building prediction models, and to possibly reduce the load on the system under monitoring if the initial number of parameters used to build the model is extremely high.

The exact parameters selected by Lasso when $\lambda = 10^9$ (i.e., the minimum number of parameters) and the weights associated with them is shown in Table 1.

`mem used slope`	-0.001349712778213
`mem free slope`	0.001349712778211
`swap used slope`	-0.000602043031343
`swap free slope`	0.000602043031343
`mem free`	0.000334016698914
`mem buffers`	0.025507773187865

Table 1: Weights assigned by Lasso

As we have mentioned before, the Framework allows the end user to select the actual way the final models are built. In fact, as we have already discussed, after Lasso has been used to regularize aggregated data points, the user can choose whether the other models should be

built using all the features, or only the ones for which Lasso selected a non-zero parameter. While we have already discussed the implications for the user of this choice, for the sake of clarity we report here final results for both execution modes, in order to effectively evaluate the effect on prediction accuracy. Of course, the results for Lasso are the same for both scenarios, as Lasso *always* selects an increasingly reduced number of parameters

3.2 Maximum Absolute Prediction Error

This error represents the highest error encountered during the prediction. It is expressed in seconds, and the corresponding values are reported in Table 2. This kind of error allows to evaluate in the worst case what could be the model' s prediction error and acts as an upper bound to assess its accuracy.

Using all parameters		**Using only parameters selected by Lasso**	
Algorithm	**Error (seconds)**	**Algorithm**	**Error (seconds)**
Linear Regression	1029.293	Linear Regression	3678.695
M5P	1038.523	M5P	997.917
REP Tree	1013.951	REP Tree	936.998
SVM	5561.302	SVM	10079.337
SVM2	5559.618	SVM2	10048.901
Lasso ($\lambda = 1$)	1179.365	Lasso ($\lambda = 1$)	1179.364
Lasso ($\lambda = 10$)	1179.362	Lasso ($\lambda = 10$)	1179.361
Lasso ($\lambda = 10^2$)	1179.336	Lasso ($\lambda = 10^2$)	1179.336
Lasso ($\lambda = 10^3$)	1179.086	Lasso ($\lambda = 10^3$)	1179.086
Lasso ($\lambda = 10^4$)	1176.814	Lasso ($\lambda = 10^4$)	1176.814
Lasso ($\lambda = 10^5$)	1171.681	Lasso ($\lambda = 10^5$)	1171.680
Lasso ($\lambda = 10^6$)	1200.604	Lasso ($\lambda = 10^6$)	1200.604
Lasso ($\lambda = 10^7$)	2847.760	Lasso ($\lambda = 10^7$)	2847.759
Lasso ($\lambda = 10^8$)	3113.035	Lasso ($\lambda = 10^8$)	3113.035
Lasso ($\lambda = 10^9$)	2516.471	Lasso ($\lambda = 10^9$)	2516.471

Table 2: Maximum Absolute Prediction Error

3.3 Relative Absolute Prediction Error

The relative absolute error is relative to a simple predictor, which is just the average of the actual values. In this case the error is just the total absolute error instead of the total squared error. Thus, the relative absolute error takes the total absolute error and normalizes it by dividing by the total absolute error of the simple predictor.

It is expressed in percentage, and the corresponding values are reported in Table 3.

Using all parameters		Using only parameters selected by Lasso	
Algorithm	**Error (percentage)**	**Algorithm**	**Error (percentage)**
Linear Regression	57.701%	Linear Regression	65.362%
M5P	34.856%	M5P	50.035%
REP Tree	31.465%	REP Tree	46.305%
SVM	55.601%	SVM	61.253%
SVM2	55.601%	SVM2	61.262%
Lasso ($\lambda = 1$)	165.580%	Lasso ($\lambda = 1$)	165.580%
Lasso ($\lambda = 10$)	165.580%	Lasso ($\lambda = 10$)	165.580%
Lasso ($\lambda = 10^2$)	165.580%	Lasso ($\lambda = 10^2$)	165.580%
Lasso ($\lambda = 10^3$)	165.576%	Lasso ($\lambda = 10^3$)	165.576%
Lasso ($\lambda = 10^4$)	165.560%	Lasso ($\lambda = 10^4$)	165.560%
Lasso ($\lambda = 10^5$)	165.424%	Lasso ($\lambda = 10^5$)	165.424%
Lasso ($\lambda = 10^6$)	165.108%	Lasso ($\lambda = 10^6$)	165.108%
Lasso ($\lambda = 10^7$)	163.062%	Lasso ($\lambda = 10^7$)	163.062%
Lasso ($\lambda = 10^8$)	163.140%	Lasso ($\lambda = 10^8$)	163.140%
Lasso ($\lambda = 10^9$)	160.381%	Lasso ($\lambda = 10^9$)	160.381%

Table 3: Relative Absolute Prediction Error

3.4 Mean Absolute Error

The Mean Absolute Error is the average of the differences between predicted and real remaining time to failure. It is expressed in seconds, and the corresponding values are reported in Table 4.

Using all parameters		Using only parameters selected by Lasso	
Algorithm	**Error (seconds)**	**Algorithm**	**Error (seconds)**
Linear Regression	141.336	Linear Regression	160.099
M5P	85.378	M5P	122.558
REP Tree	77.071	REP Tree	113.421
SVM	136.192	SVM	150.035
SVM2	136.191	SVM2	150.058
Lasso ($\lambda = 1$)	405.221	Lasso ($\lambda = 1$)	405.221
Lasso ($\lambda = 10$)	405.221	Lasso ($\lambda = 10$)	405.221

Lasso ($\lambda = 10^2$)	405.220	Lasso ($\lambda = 10^2$)	405.220
Lasso ($\lambda = 10^3$)	405.212	Lasso ($\lambda = 10^3$)	405.212
Lasso ($\lambda = 10^4$)	405.148	Lasso ($\lambda = 10^4$)	405.148
Lasso ($\lambda = 10^5$)	404.839	Lasso ($\lambda = 10^5$)	404.839
Lasso ($\lambda = 10^6$)	404.066	Lasso ($\lambda = 10^6$)	404.066
Lasso ($\lambda = 10^7$)	399.059	Lasso ($\lambda = 10^7$)	399.059
Lasso ($\lambda = 10^8$)	399.250	Lasso ($\lambda = 10^8$)	399.250
Lasso ($\lambda = 10^9$)	329.497	Lasso ($\lambda = 10^9$)	329.497

Table 4: Mean Absolute Error

3.5 Soft-Mean Absolute Error

The Soft-Mean Absolute Error is calculated as the Mean Absolute Error except that when the predicted value is below a given threshold, it is assumed to be equal to 0.

Tolerance threshold: 10%

In this case, if the prediction is less than 10% of the real value, then the prediction is assumed to be equal 0. In Table 5, values are given in seconds.

Using all parameters		**Using only parameters selected by Lasso**	
Algorithm	**Error (seconds)**	**Algorithm**	**Error (seconds)**
Linear Regression	137.600	Linear Regression	156.603
M5P	79.182	M5P	118.292
REP Tree	69.832	REP Tree	108.476
SVM	132.668	SVM	146.594
SVM2	132.675	SVM2	146.607
Lasso ($\lambda = 1$)	405.187	Lasso ($\lambda = 1$)	405.187
Lasso ($\lambda = 10$)	405.187	Lasso ($\lambda = 10$)	405.187
Lasso ($\lambda = 10^2$)	405.186	Lasso ($\lambda = 10^2$)	405.186
Lasso ($\lambda = 10^3$)	405.178	Lasso ($\lambda = 10^3$)	405.178
Lasso ($\lambda = 10^4$)	405.124	Lasso ($\lambda = 10^4$)	405.124
Lasso ($\lambda = 10^5$)	404.823	Lasso ($\lambda = 10^5$)	404.823
Lasso ($\lambda = 10^6$)	404.041	Lasso ($\lambda = 10^6$)	404.041
Lasso ($\lambda = 10^7$)	399.023	Lasso ($\lambda = 10^7$)	399.023
Lasso ($\lambda = 10^8$)	399.240	Lasso ($\lambda = 10^8$)	399.240

Lasso ($\lambda = 10^9$)	392.469	Lasso ($\lambda = 10^9$)	392.469

Table 5: Soft-Mean Absolute Error (10% tolerance, memory leaks)

Tolerance threshold: 5 minutes (300 seconds)

In this case, if the prediction is less than 300 seconds, then it is assumed to be equal 0. In the following table, values are given in seconds.

Using all parameters		**Using only parameters selected by Lasso**	
Algorithm	**Error (seconds)**	**Algorithm**	**Error (seconds)**
Linear Regression	51.790	Linear Regression	73.538
M5P	20.218	M5P	35.460
REP Tree	14.969	REP Tree	32.901
SVM	53.288	SVM	66.424
SVM2	53.289	SVM2	66.406
Lasso ($\lambda = 1$)	345.477	Lasso ($\lambda = 1$)	345.477
Lasso ($\lambda = 10$)	345.477	Lasso ($\lambda = 10$)	345.477
Lasso ($\lambda = 10^2$)	345.475	Lasso ($\lambda = 10^2$)	345.475
Lasso ($\lambda = 10^3$)	345.612	Lasso ($\lambda = 10^3$)	345.612
Lasso ($\lambda = 10^4$)	345.595	Lasso ($\lambda = 10^4$)	345.595
Lasso ($\lambda = 10^5$)	345.335	Lasso ($\lambda = 10^5$)	345.335
Lasso ($\lambda = 10^6$)	345.475	Lasso ($\lambda = 10^6$)	345.475
Lasso ($\lambda = 10^7$)	346.709	Lasso ($\lambda = 10^7$)	346.709
Lasso ($\lambda = 10^8$)	355.474	Lasso ($\lambda = 10^8$)	355.474
Lasso ($\lambda = 10^9$)	335.287	Lasso ($\lambda = 10^9$)	335.287

Table 6: Soft-Mean Absolute Error (5 minutes tolerance, memory leaks)

3.6 Training Time for ML models with WEKA (1 instance of the model)

The Training Time represents the time taken to instantiate a prediction model from the input dataset. It is expressed in seconds in Table 7.

Using all parameters		**Using only parameters selected by Lasso**	
Algorithm	**Time (seconds)**	**Algorithm**	**Time (seconds)**
Linear Regression	0.30	Linear Regression	0.08

M5P	3.10	M5P	1.58
REP Tree	0.56	REP Tree	0.17
SVM	417.41	SVM	164.96
SVM2	391.69	SVM2	205.65

Table 7: WEKA Training Time (memory leaks)

3.7 Validation Time

The Validation Time represents the time needed to validate the accuracy of a model. It is expressed in seconds in Table 8.

Using all parameters		**Using only parameters selected by Lasso**	
Algorithm	**Time (seconds)**	**Algorithm**	**Time (seconds)**
Linear Regression	0.42	Linear Regression	0.12
M5P	0.36	M5P	0.09
REP Tree	0.55	REP Tree	0.11
SVM	0.39	SVM	0.13
SVM2	0.38	SVM2	0.13

Table 8: WEKA Validation Time (memory leaks)

3.8 Fitted Models

We report in the following Figure 11-Figure 20 the fitted models for $\lambda = 10^9$ using all the WEKA training models plus Lasso in Figure 21. In the plots, the green line is the ground truth, while the red curves express the predicted Remaining Time to Crash by the involved model, for the selected value of λ.

In the plots, on the x-axis we have the RTTC/RTTH, while on the y-axis we have the *predicted* RTTC/RTTH. As mentioned before, the Framework does not distinguish between RTTC/RTTH because the training is done taking into account memory leaks, unterminated threads or both. Therefore, since we are only interested in what is the best suited instant to rejuvenate the system, it does not make any difference whether we have to rejuvenate because of a crash due to memory leaks or because of a threshold hit due to unterminated threads. Hence, we refer to both as RTTC/RTTH.

We note that Lasso (with $\lambda = 10^9$) in Figure 21 is the only configuration where no false positives are encountered.

Using all parameters

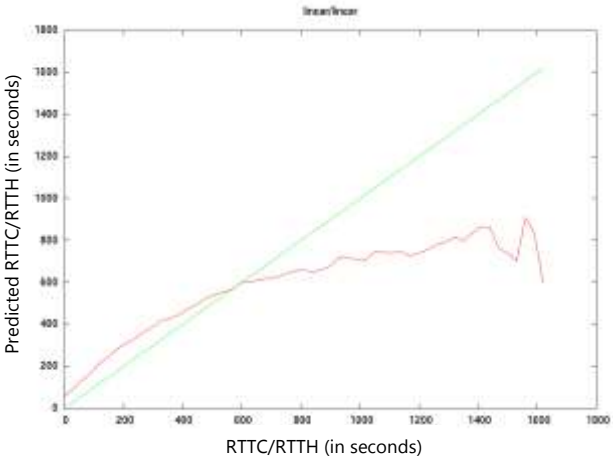

Figure 11: Prediction with Linear Regression

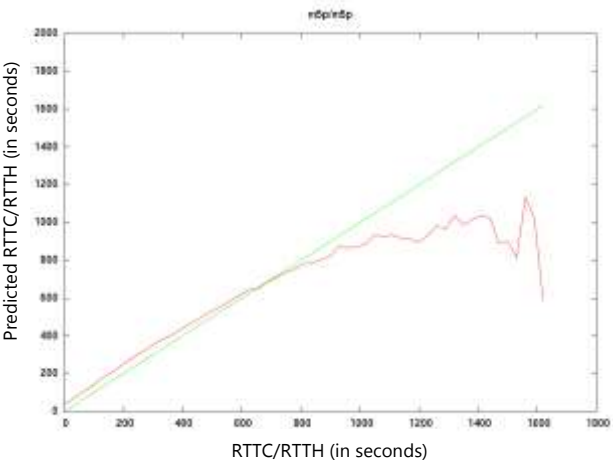

Figure 12: Prediction with MP5

Using only parameters selected by Lasso

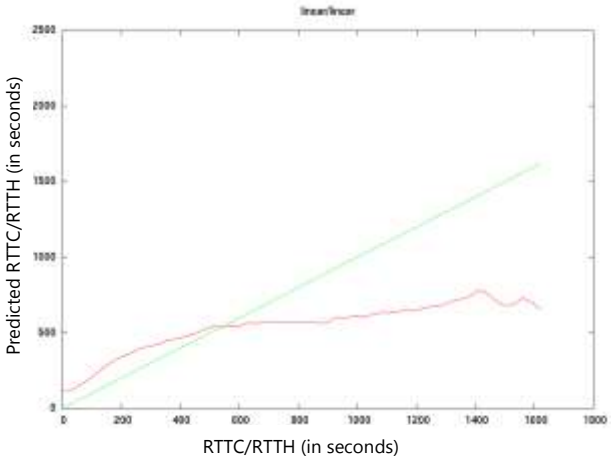

Figure 16: Prediction with Linear Regression

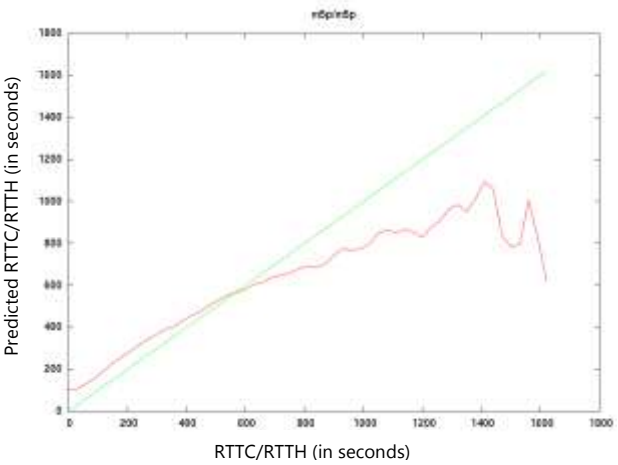

Figure 17: Prediction with MP5

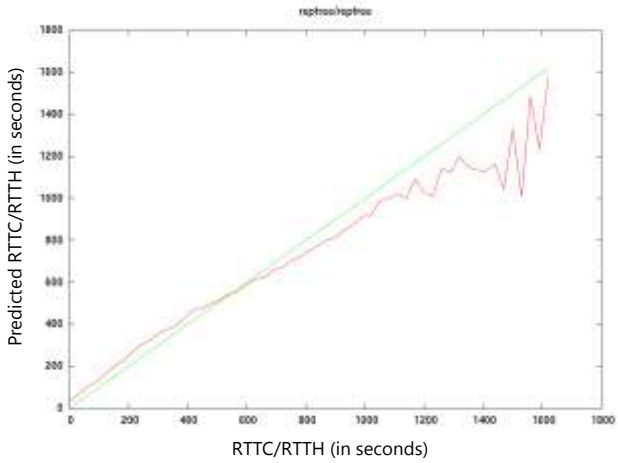

Figure 13: Prediction with REP Tree

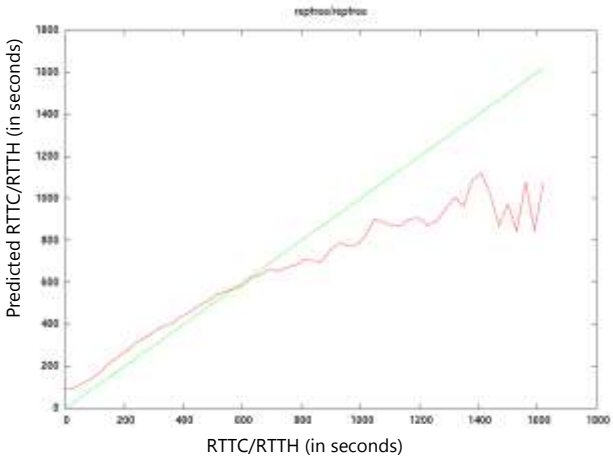

Figure 18: Prediction with REP Tree

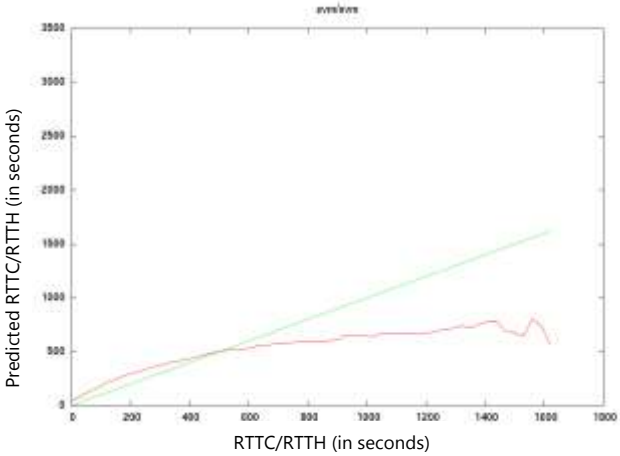

Figure 14: Prediction with SVM

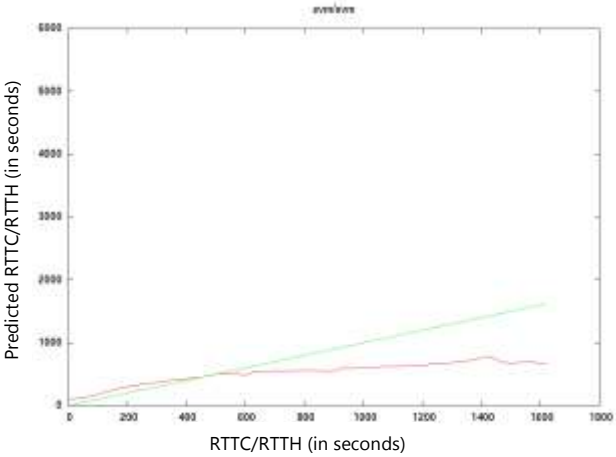

Figure 19: Prediction with SVM

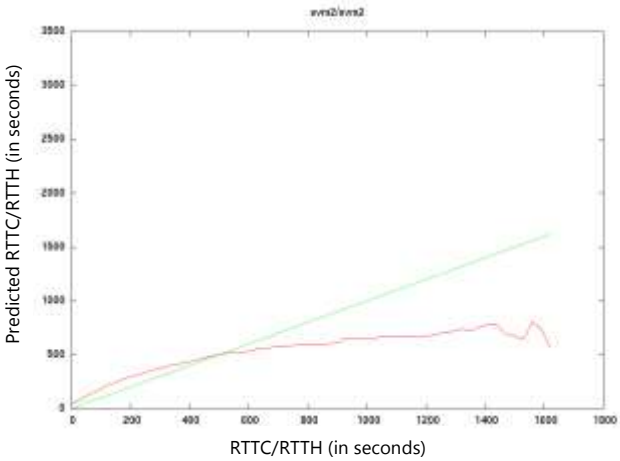

Figure 15: Prediction with SVM2

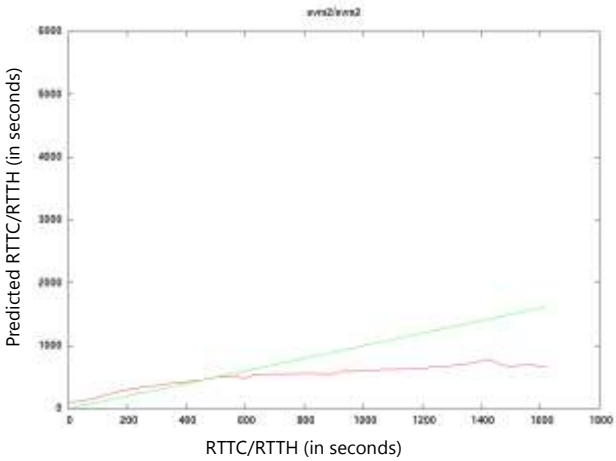

Figure 20: Prediction with SVM2

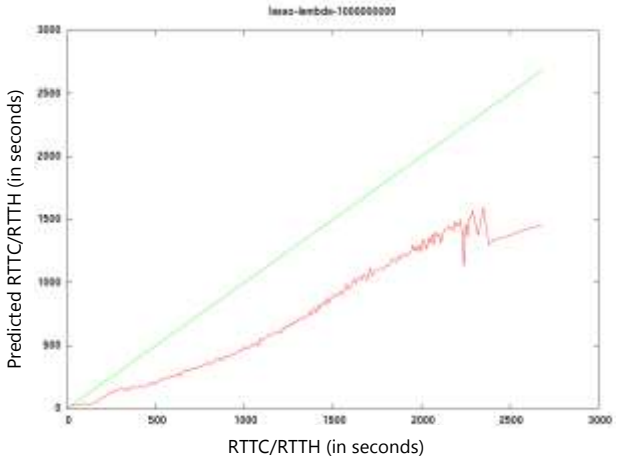

Figure 21: Prediction with Lasso as a Predictor ($\lambda = 10^9$)

3.9 Response Time variation

In Deliverable 3.1 [14] we have shown how the Response Time seen by clients changed when using the Proactive Rejuvenation mechanism. For the sake of completeness, we report here the results seen by the same configuration of the system as described in Deliverable 3.1 [14], using the newly presented prediction model.

The RTTC/RTTH and the System Features, in the presence memory leaks and unterminated threads, are presented in Figure 22. The red lines represent points in time when one of the two virtual machines are rejuvenated. The rejuvenation is related to either an approaching Crash point, or because the (correlated) RT is reaching the Threshold discussed in Section 2, which has been set to 4 seconds.

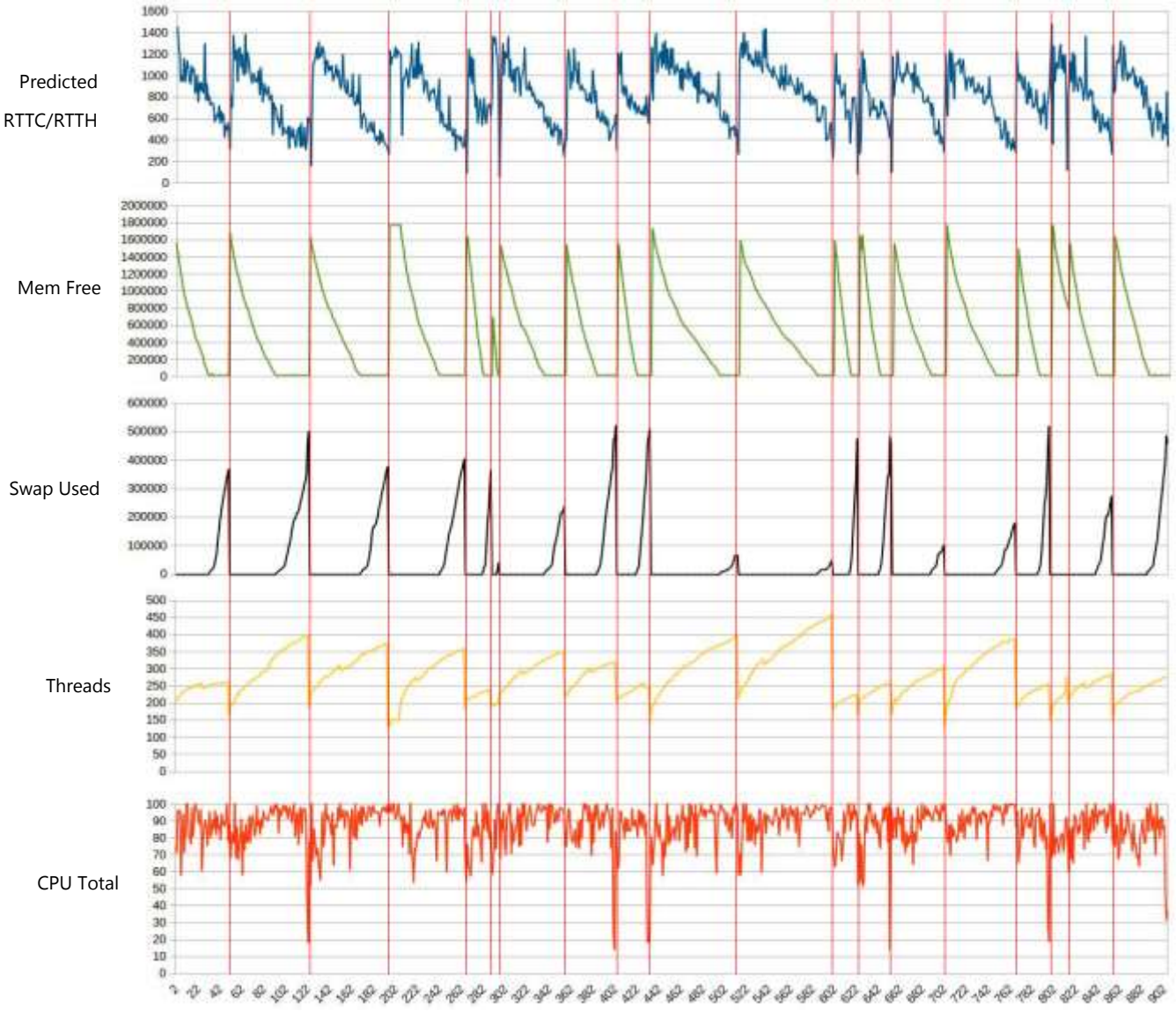

Figure 22: Average System Response Time and System Features with rejuvenation (leaks and threads)

Figure 23: Response Time (leaks and threads)—TPC-W Execute Search Interaction

Figure 23 shows the average RT for the TPC-W *Execute Search* web interaction. Red lines are associated with points in the execution time where the machine, to which the Clients were

connected, was given the rejuvenate command. It was sent by the controller either because the VM was approaching the crash point, or because the (correlated) RT was nearly to violate the threshold.

In some cases (see, e.g., the first rejuvenation), the clients experience a slower decrease in RT. This is related to the fact that when one VM has to rejuvenate, it enters first the Finish Requests state (see Figure 7). In this context, the TPC-W Users (which are at the same time issuing new requests, which are immediately forwarded to the newly Active machine) experience a different response time, depending on which (among the two running VMs) is servicing each request. Therefore, there is a time window in which the RT can decrease more gradually, due to the fact that two VMs are actually concurrently running.

3.10 Discussion of the results

ML Framework generates predictions models with all features (monitored parameters of the physical resources) or with a reduced set of parameters, based on Lasso Regularization technique. The users can make a choice, based on the operational requirements of their cloud applications.

For applications where false negatives are not allowed the reduced set of parameters must be selected by the users. The Lasso clearly demonstrates that the selection time for rejuvenation can guarantee the absence of false negatives and for a large number of monitored parameters it is the preferable choice.

For smaller number of parameters and if the false negatives are allowed, the variant with all parameters might be selected, given that they provide smaller prediction error.

4 EXPERIMENTAL ENVIRONMENT AT THE NODE LEVEL

The goal of this research is to show the effectiveness of a Global Architecture for Proactive Management in a (geographically) distributed environment. While the distributed architecture will be presented in Section 5, we discuss here the software used on each VM implementing a copy of the web server. This will allow to highlight the software configuration in what the users are really interested in using and seeing, while the description of the elements of the architecture aiming at providing enhanced availability via proactive management (rejuvenation) is in Section 5.

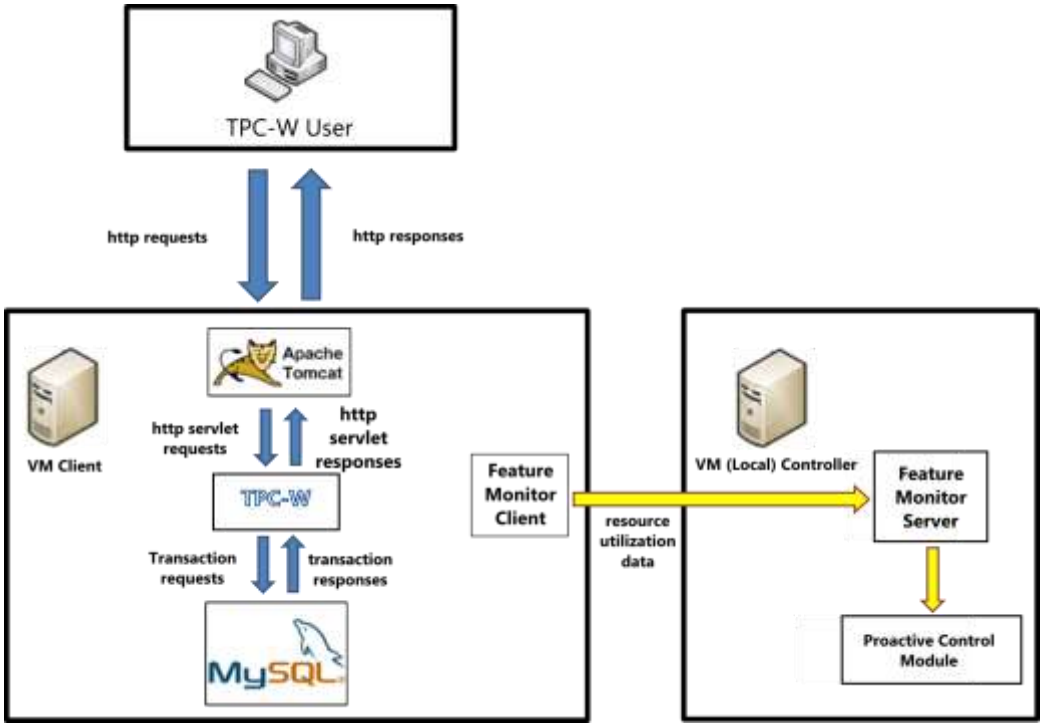

Figure 24: Software Configuration at Node Level and Local Controller

The general node-level architecture is shown in Figure 24. Each web server hosts the TPC-W Web Application standard benchmark [21], which requires several pieces of software to run correctly:

- Apache Tomcat 6.0.41 [22]
- A Java implementation of TPC-W [21]
- MySql Server version: 5.1.41-3 [23]
- A *Feature Monitor Client* , which is a proprietary software package aimed at collecting resource usage statistics of the VM Client machine and send it to the Feature Monitor Server

The Java implementation of TPC-W relies on Java HTTP Servlet, a standard for implementing java classes for handling HTTP requests, allowing to manage dynamic contents in a web

server through the Java platform. Multiple TPC-W users, implemented as emulated browsers (see Deliverable 3.1 [14] for a complete description of the behaviour of the clients) are used to simulate the traffic generated by end users. Given the fact that TPC-W Clients can be (geographically) distributed, and provided that their location is of no constraint to the generality of our approach, they are depicted as a single box in Figure 24.

The TPC-W workload configuration parameters are: the number of users; the client average think time (time between the http response and subsequent http request); the workload interaction mix (type and percentage of interactions executed by users). While we have used one single interaction mix for our experimentation (see again Deliverable 3.1 [14]), the number of TPC-W Clients—as already mentioned—has been varied in between 8 and 128 during the training phase carried out for this deliverable, and already discussed in Section 3. On the other hand, during the experimentation of the actual (distributed) rejuvenation architecture , we have set this parameter to 64.

In fact, Figure 25 reports the RT as seen by the Clients when no anomalies where injected. The plot shows the Inverse Cumulative Function, along with confidence intervals. This function tells (for a given time value on the x axis in seconds) what is the amount (in percentage) of web interactions that take more than that amount of time to complete.

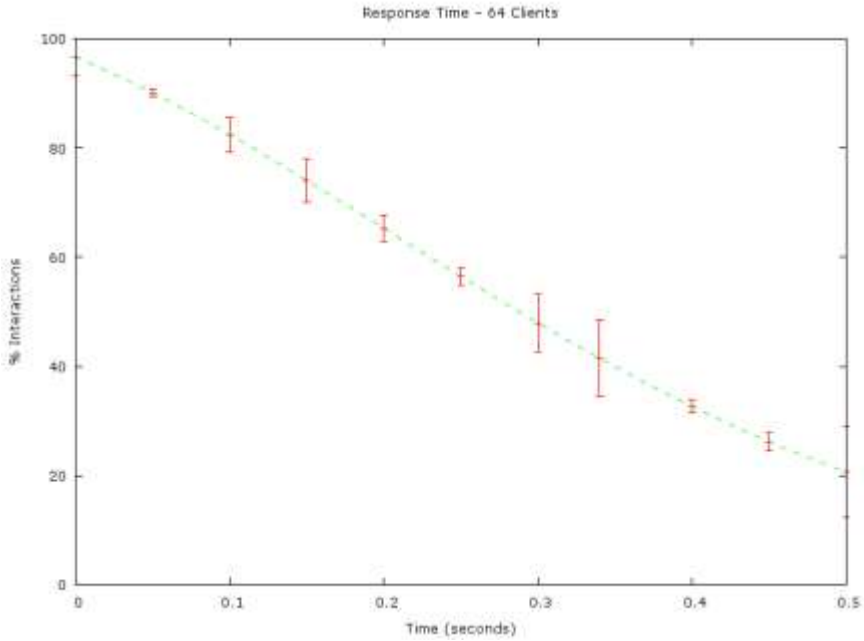

Figure 25: TPC-W Response time with 64 Users

It is important to emphasize that we have selected as the plotting range for the x axis the interval [0, 0.5]. In fact, we consider 0.5 seconds to be still a sustainable response time for application users who are using the web application, since it allows to have anyhow timely responses. If the green line converges quickly to zero, the system is not overloaded. Using 64 Clients is motivated by the fact that in this configuration the workload is non-minimal (80% of web interactions require 0.5 seconds or less to complete), yet the system is not yet thrashing, therefore allowing us to capture the dynamics of injected anomalies independently of anomalies created by a too-high workload generated by the application.

The goal of the Feature Monitor Client is similar in spirit to that of the data collection phase. In fact, the Feature Monitor Client collects resource utilization data of the VM client and sends them to the Feature Monitor Server. The Feature Monitor Server is installed on a different virtual machine, where the actual controller (targeted at proactively deciding for the rejuvenation of VMs) is running.

As it will be thoroughly described in Section 5, multiple controllers are used when dealing with a distributed virtualized system. For the sake of simplicity, let us assume now that only two VMs are present, one running TPC-W Server, and the other running the controller. In this scenario, since the controller can be uniquely identified in the network (e.g., by relying on standard IP addresses), features can be passed to the proactive control module for prompt failure prediction.

Specifically, within the Proactive Control Module installed on the controller VM, all the prediction models described in Section 3 are available, and upon its startup the user/system administrator is able to select which one should be evaluated upon the receipt of one new set of hardware features. In this way, when on VM Client the occurrence of anomalies alters the measured features, VM Controller is able to correctly predict the RTTC/RTTH of VM Client.

We note that, by relying on this simple (yet effective) architecture, we are able as well to indirectly predict the RT as seen by TPC-W Users. In fact, VM Controller uses a model which was built using as well additional added derived metrics, among which the Generation Time of datapoints was present. Then, by the discussion in Section 2, we already know that this information can be correlated with the actual RT (ground truth) seen by TPC-W Users, without any need for actually measuring it, a task which could be difficult if not impossible—in fact TPC-W Users might be (geographically) distributed, and it would be practically impossible to keep track of all of them, not mentioning the intrusiveness of the insertion of software probes to gather this data.

Actually, more than one VM Controller could be present. In fact, given that we are explicitly targeting distributed rejuvenation, theoretically speaking multiple sets of VMs (hosted on different physical hosts) could rely on multiple local controllers. Even more generally speaking, when dealing with overlay networks (where multiple geographically-distributed VMs share a virtualized local network), multiple virtual clusters of VMs could rely on one single (virtually) local controller.

This flexibility of the approach comes directly from the simplicity of the node-level architecture, as described in Figure 24. In fact, since we are only relying on a client/server architecture, it is possible to add virtually any additional communication layer to allow for the interaction of VMs with their respective local controller, independently of the physical location of the VM. This flexibility will be explicitly exploited in the remainder of this experimentation, as per the description in Section 5.

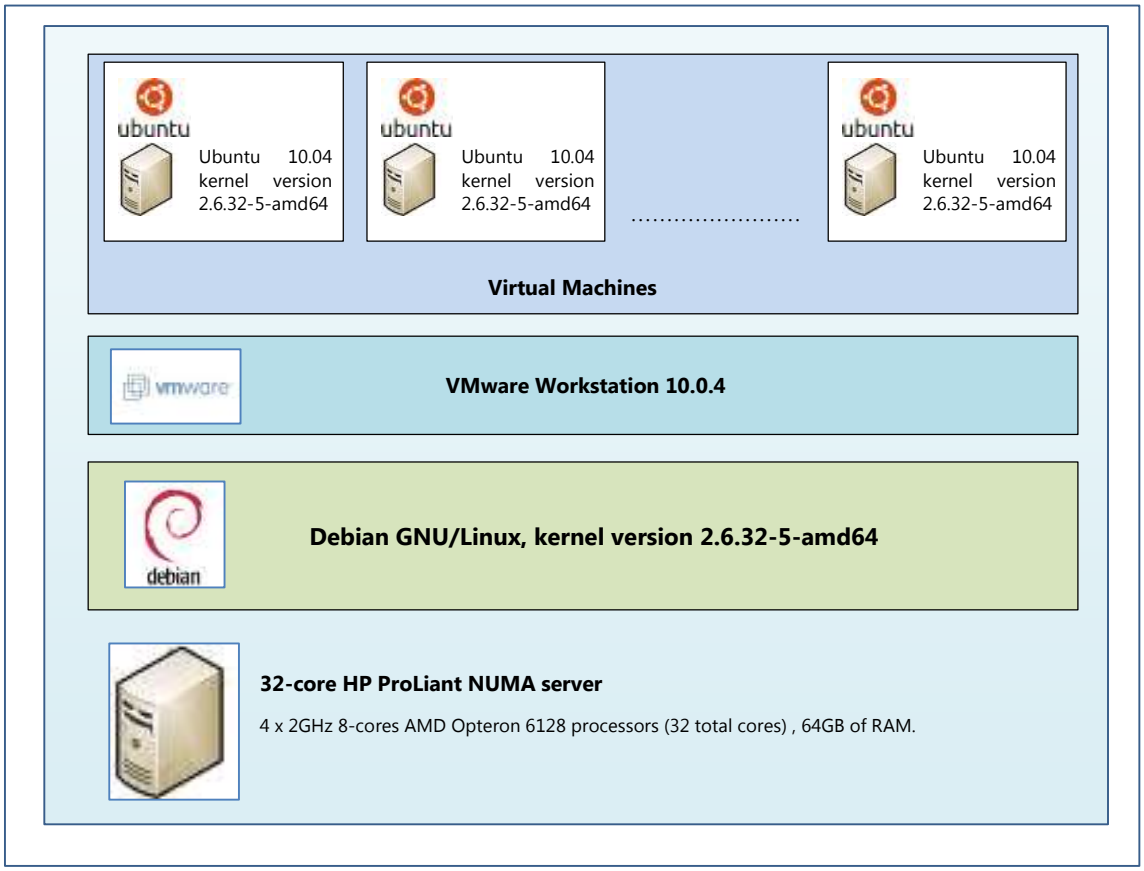

Figure 26: Experimental Virtual Environment (one example hardware host)

Concerning the single hardware host on which VMs are installed, it is structured as in Figure 26. It consists of a virtual architecture, which is built on top of a 32-core HP ProLiant NUMA server. The server is equipped with a Debian GNU/Linux distribution (kernel version 2.6.32-5-amd64). VMware Workstation 10.0.4 is the virtual environment hypervisor. All virtual machines of the experimental environment are equipped with Ubuntu 10.04 Linux Distribution (kernel version 2.6.32-5-amd64). In the case of the distributed environment described in Section 5, multiple *heterogeneous* hardware hosts will be used, distributed in Europe.

5 DISTRIBUTED ARCHITECTURE FOR PROACTIVE MANAGEMENT

The Distributed Architecture for Proactive Management implements an autonomic system [24], which relies on an Autonomic Manager (AM) to detect whether any VM in the distributed system is about to crash and enforce prompt operations to reduce at most the effect of this crash on the users of the system. The overall general architecture of the elements of the proactive system are given in Figure 27.

It is interesting to emphasize that the same architecture described in Figure 27 can be used in a twofold nature, as already mentioned before:

a) Intelligent collection of Training Data and usage of Machine Learning algorithms for creating prediction models for proactive management of Intra cloud resources. Intra ACM is communicating with other Intra ACM and Main Controller (Inter Autonomic Cloud Manager), as later described.

b) Local Controller (Intra Autonomic Cloud Manager) does not create Training Data Set. On the other hand, it decides and generates control signals that trigger the proactive management of cloud resources based on the inputs from the Feature Monitor Client(s) (Figure 24).

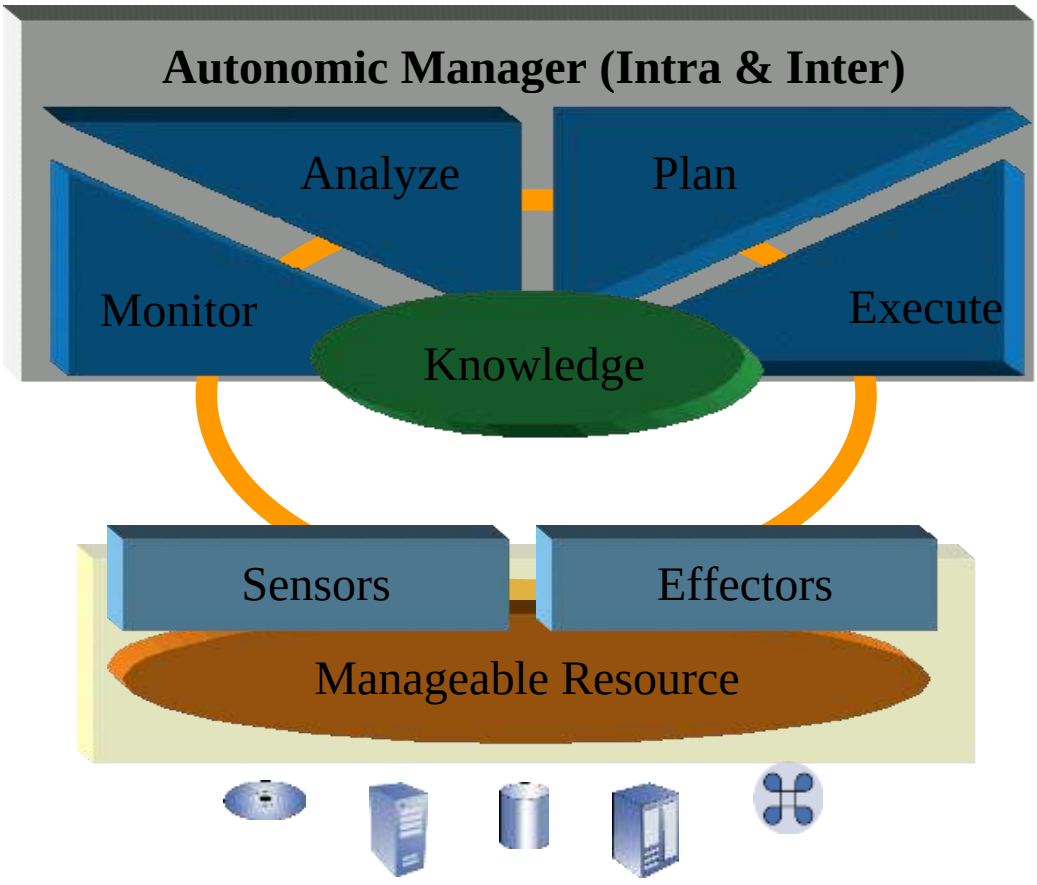

Figure 27: Architecture Elements: Autonomic Manager, Manageable Resources and Knowledge Resources

The same architecture in Figure 27, implemented within the Local Controller, is used in a twofold nature, namely an On-Line and an Off-Line mode. These modes of operations of the Inter/Intra–Autonomic Cloud Managers are presented for the realization of a control loop in their operations and Proactive Management of cloud resources. ML Framework & Global Architecture for Proactive Management are defined and the basic modules are implemented. Intra-ACMs forming Private Clouds and communicating via Inter-ACMs are capable of creating Federated Clouds

As already mentioned in Section 3, the Knowledge base is created from Training Data Base using the ML Framework, which generates ML prediction models. The chain of Manageable Resources (VMs installed in HP servers or geographically via Internet), Sensors and Monitor are used to construct the Knowledge Resources. Each Intra-ACM collects the features (physical parameters during the application execution), builds Data Base and utilizes the ML Framework or sends them to the Inter-ACM for forming the Knowledge Resources. The ML Framework is activated in order to analyse the Training Data Base and produce the Prediction Models for proactive management of cloud resources controlled by Intra-ACM/Inter-ACM (Plan module.) Dynamic Reconfiguration Flow Diagram based on ML prediction is activated by the Execute and Effectors. In this way, the control loop is closed and the Manageable Resources again are censored and monitored for performing proactive autonomic management of cloud resources. Based on the presented features of the Autonomic Manager (Inter/Intra) control loop the proactive management of the cloud resources can be accomplished and autonomic self* management can be achieved.

The specified modules in Figure 27 realize different functions in each Off-Line or On-Line mode of operations of Autonomic cloud Managers.

There are two levels for creating ACMs:

1) The first level (OFF-Line ACM) activates *Sensors* and *Monitor* modules to collect the features by Feature Monitor Client under anomalies (memory leaks and unterminated threads) in the Manageable Resources. Training Data base is created in the unit *Analyze*. Important decision are made by the module *Plan*. For example, determining the Remaining Time to Crash (RTTC) and Threshold for the Response time of web servers or Execution time of cloud applications, Web applications Availability requirements. The *Execute* segment determines the set of Machine Learning Algorithms that will be used for creating the ML Prediction Models and techniques for reducing the number of monitoring parameters (Lasso regularization), tools for automatically generating prediction models and defining the Machine Learning Framework. The pool of (joining and leaving) VMs is created and their state (active, standby) is determined in the *Effectors* unit. The unit "Effectors" starts the operation of ML Framework & Global Architecture for Proactive Autonomic Management.
2) In the second Level (On-Line ACM) *Sensors and Monitor* modules collect features (parameters) under injected anomalies from the Manageable Resources. There are two options: All parameters collected in Off-Line level or Reduced Number of Parameters determined by Lasso techniques. The module *Analyse* evaluates the RTTC/RTTH of Web servers. An important function of *PLAN* unit is compute the difference between the run time predicted RTTC/RTTH and the selected interval for safe rejuvenation of corresponding active VM. The same steps are taking by the *PLAN*

unit for run time predicted threshold of the response (execution) time. The generated signal by PLAN unit will allow *Execute* to activate *Effectors* module. In this case, *Effectors* module triggers the safe rejuvenation step in the Dynamic Rejuvenation Flow diagram. If this signal is not generated then the control loop continues through all steps until the *Effectors* module is activated.

Figure 28 shows the architecture of the distributed machine learning framework. There are two locations (later be extended) with eight virtual machines on each location. Some virtual machines are managed by the Main Controller placed in the Local Virtual Infrastructure, while other sets of virtual machines are managed by Local Controllers in either Local Virtual Infrastructures or in Hybrid Clouds. The Local Controllers are in charge of collecting data from managed virtual machines. They use this collected data to evaluate the prediction models. Then, the Local Controllers send to the Main Controller information about the current load of the managed VMs through the Overlay Network (Transfer Agent). The Main Controller collects data from virtual machines in its Local Virtual Infrastructure as well, and creates the ML framework. It implements as well a global load balancer using the load information sent by Local Controllers to orchestrate the overall set of requests by the end users via load balancing. Rejuvenation actions for virtual machines are generated both by Main Controller (for the locally-managed VMs) and by the Local Controllers (for the locally-managed VMs). In both controllers (Main and Local) load balancer techniques are used to distribute the load between virtual machines.

Agents are used for creating overlay topologies (between two controllers) for selecting minimum paths and delays between them. TCP-IP protocol or UDP protocol can be used for communications.

In the case of Hybrid Cloud, a Local Controller is able to communicate with locally-hosted virtual machines using local IP addresses, while it can communicate at the same time with VMs in the Cloud using their public IP addresses. The only prerequisite is that all VM Clients, as shown in Figure 24, both locally-hosted and in the Cloud have installed the Feature Monitor Client, as described in Section 2.

The Overlay network is composed of several Intra overlays that are interconnected by an Inter overlay network, as described in Figure 28. In this task, we will mainly focus on application-specific inter-cloud overlays. The proposed overlay support will use the machine learning algorithms, developed in WP3, in order to proactively reconfigure the application-specific overlays when an adverse event is predicted. It will also be able to derive the most adequate topology from the application requirements given by the user. To this end, we will have to solve the following problems: Topology creation and reconfiguration; Dynamic routing within the overlay network; Resiliency to partitioning.

There are two cases for set up the VMs, as it is pointed out in Section 2.3:

a) Installing all VMs in HP Server, controlled by the VMware hypervisor (left down Local Controller and left up Main Controller in Private Local Virtual Infrastructure in Figure 28).

b) Mounting a portion of VMs in HP Server and controlled by the VMware hypervisor. The rest of VMs are placed in the Cloud (right down and right up Local Controller of

Hybrid Cloud[2] in Figure 28). In the future, VMs, independently of their location, could be allowed to communicate via Overlay Networks.

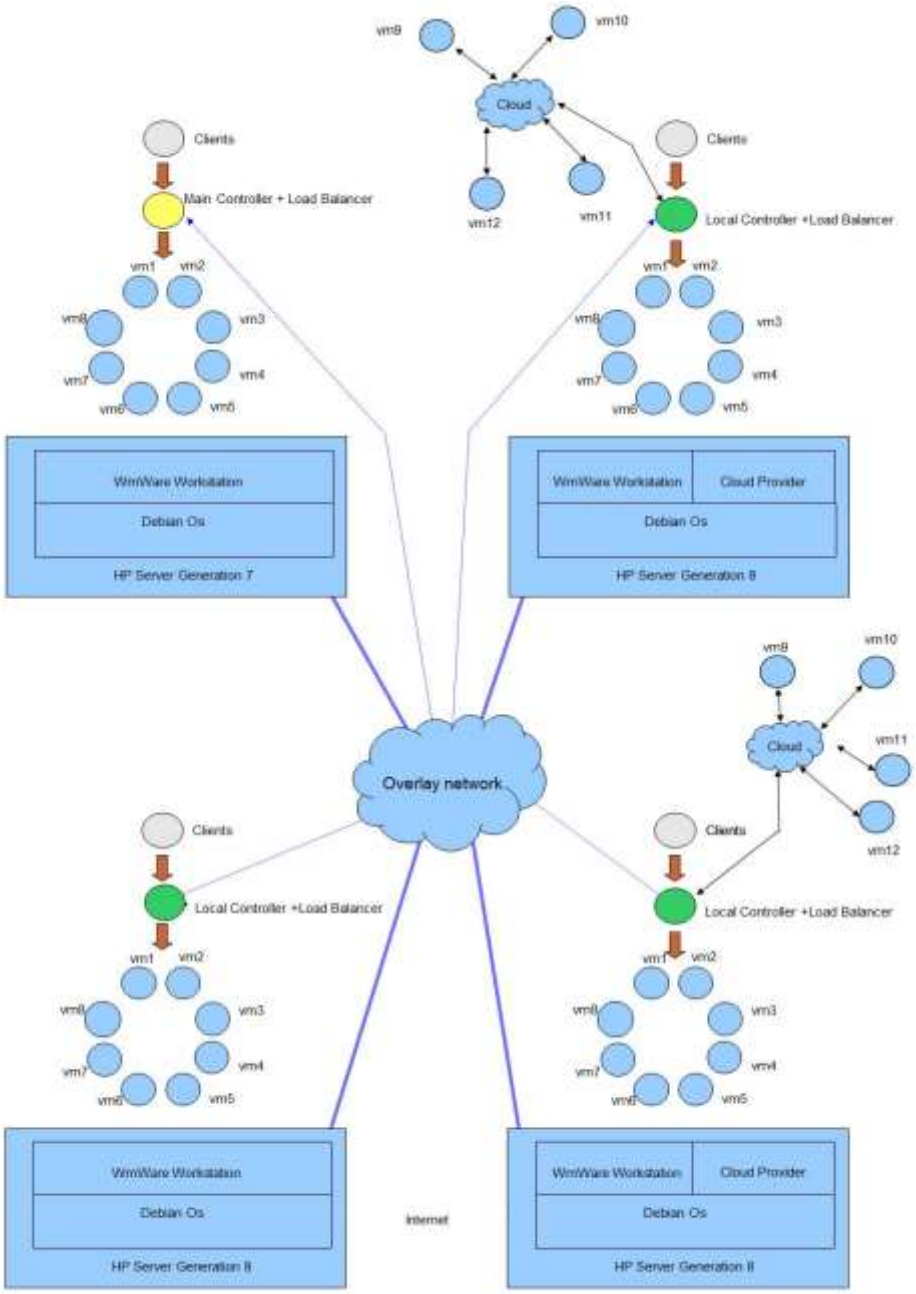

Figure 28: ML Framework and Global Architecture for Proactive Management

[2] A hybrid cloud is a cloud computing environment in which an organization provides and manages some resources in-house (in our case, the multicore server) and has others provided externally (e.g., by Cloud Provider). Hybrid Clouds are necessary when dealing with critical data (in a portion of the Hybrid Cloud, in the HP Servers, such as maintaining large database for prediction models.), which is an applicable scenario for our Framework.

All the HP Servers are protected by Firewalls, so as to allow for secure storage of the critical data hosted in the Hybrid Clouds.

In Figure 29, the virtual network topology is shown. The network includes 8 nodes. Each node is connected with other 4 nodes in the network. The virtual topology is scalable (Up and Down) and resilient to partitioning in presence of multiple node and links failures.

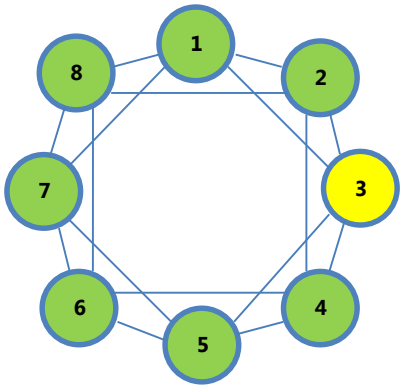

Figure 29: Virtual Network Topology with 8 Nodes

VM3 is the Inter-ACM (Main Controller 3-yellow in Figure 28 and Figure 29), which can talk to other controllers Intra-ACM (1, 2, 4, 5, 6, 7, 8-green) in the virtual topology, where Node 2 (left down Local Controller in Private Local Virtual Infrastructure in Figure 28), Node 4 (right down Local Controller of Hybrid Cloud in Figure 28), Node 5 (right up Local Controller of Hybrid Cloud in Figure 28).

The Intra-ACM and Inter-ACMs can create virtual topology for distribution of tasks and workloads among different Hybrid Clouds and Private Local Virtual Infrastructures, as presented in Figure 28.

The Inter-ACM (Main Controller) is in charge of orchestrating the Hybrid Clouds and the Private Local Virtual Infrastructures.

6 DISTRIBUTED EXPERIMENTAL ENVIRONMENT FOR PROACTIVE MANAGEMENT

In both controller (Main and Local) a load balancer distributes the load between virtual machines.

Agents are used for creating overlay topologies (between two controllers) for selecting minimum paths and delays between them. TCP-IP protocol or UDP protocol can be used for communications.

The architecture of the Autonomic Cloud Manager (ACM) is shown in Figure 28.

6.1 Experimental Results

In Deliverable 3.1 [14] and in Section 3.9 we have shown results where only two different VMs where used to respond to TPC-W Users requests, one being Active, one being Stand By. The Stand By machine was activated by the Controller upon the approaching of a system crash.

We hereby complement those results in a more complex environment. Specifically, we have run experiments for 3 consecutive days, using a set of "locally distributed" VMs. Specifically, we have realized the abstract architecture depicted in Figure 6, where one VM is the local controller, one VM hosts TPC-W Users, and the remaining 6 VMs host copies of the TPC-W Server application, as described in Figure 24.

We have grouped this 6 virtual machines in 3 couples. Each couple is subject to different injection of anomalies—namely, one couple is subject to memory leaks, one couple to unterminated threads, and the last to both memory leaks and unterminated threads. In this experimentation, the controller has been equipped with the 3 different prediction models, the two from Deliverable 3.1 [14] for memory leaks and unterminated threads, and the one described in Section 3 of this deliverable for both memory leaks and unterminated threads.

Under this scenario, we are able to evaluate the resilience of the controller and the effectiveness of the approach to promptly detect whether one VM is approaching its crash point (or if the clients connected to it are reaching the RT threshold). Overall, this experiment allows us to show that TPC-W Users connected to the system are not suffering a significant increase of the RT although the VMs are actually suffering from the injection of anomalies.

At the beginning of the experiment, 3 VMs are Active, while the remaining 3 VMs are in the Stand By state. At runtime, during the execution of the experiment, more than 3 VMs can be Active at the same time, because just before the actual rejuvenation, one VM enters the Finishing Requests state, as shown in Figure 7. Under this assumption, at any time instant, any number of VMs in between 3 and 6 are allowed to be serving TPC-W Users requests.

An important note regards how TPC-W Users are connected to the VMs. As already mentioned, due to the dynamic nature of the system, it is impossible for the clients to connect directly to them. More specifically, since TPC-W Users must be unaware of the internal topology of the VMs, and due to the fact that they should know nothing about the rejuvenation process (TPC-W Users are only interested in the services offered by the VMs, not in how the VMs *do* provide these services), they cannot know what is the actual IP address of the *currently* active VM. Therefore, an important role is here played by the Load Balancer, which (as depicted in Figure 28) is installed on the same VM as the Controller.

Specifically, the Load Balancer receives request from the TPC-W Users. It then simply implements a *forwarding rule* to let the request be received from the currently active VM. At any time when the Controller rejuvenates one VM (and therefore activates a twin VM), the Load Balancer is instructed by the Controller to let *new* requests be sent to the newly activated VM. Any request already sent to the rejuvenating VM is served by that VM (which is therefore in the Finishing Requests state, as depicted in Figure 7). In this way, TPC-W Users can promptly reach the new Active VM, without the need to know anything about the internal configuration of the logical cluster of VM.

To allow for a fair experimentation, the Load Balancer has been configured to send requests to the various VMs in a round robin fashion (but any other balancing technique is supported by the load balancer). Specifically, each set of Web Interactions received from the TPC-W Users is forwarded to one of the 3 couples of VMs. In this way, we explicitly avoid to create any artificial bias in the workload sent to the couples of VMs.

In the following plots, we report data related to only few hours of the 3 days of execution of the experiment. This has been done in order to enhance readability of the results, which are nevertheless fully representative of the whole execution of the experiment (no statistical deviations have been observed in the remainder of the data).

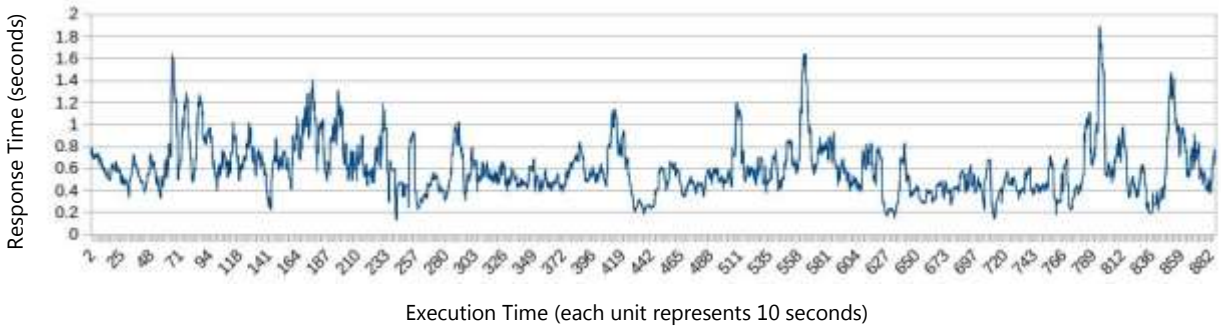

Figure 30: Average Response Time with 3 couples of VMs

In Figure 30 we present the *average* RT seen by TPC-W Users. Given the round-robin fashion of the request forwarding, and given the fact that VMs suffer from the injection of different anomalies, this plot is not representative of the behaviour of a single VM under one specific anomaly, rather it shows how the Users would see the system behaviour in an *agnostic* way, meaning that they know nothing about the internal implementation, rather they are only interested in having their requests be served quickly. While we will show in Figure 31 that the VMs were actually suffering from anomalies injection, the RT seen by the Users is kept, on average, very low. In fact, although there are some peaks in the RT (approaching 2 seconds), the RT is kept, on the average, at 0.7 seconds. This result is interesting in a twofold manner. On the one side, the average response time (0.7 seconds) is very low, allowing for an effective usage of the system by the Users. On the other side, the (average) peak (less than 2 seconds) is very far from the threshold of 4 seconds which was set during the training phase. This shows that Lasso as a predictor never suffers from false positives, so that we never exceed the threshold.

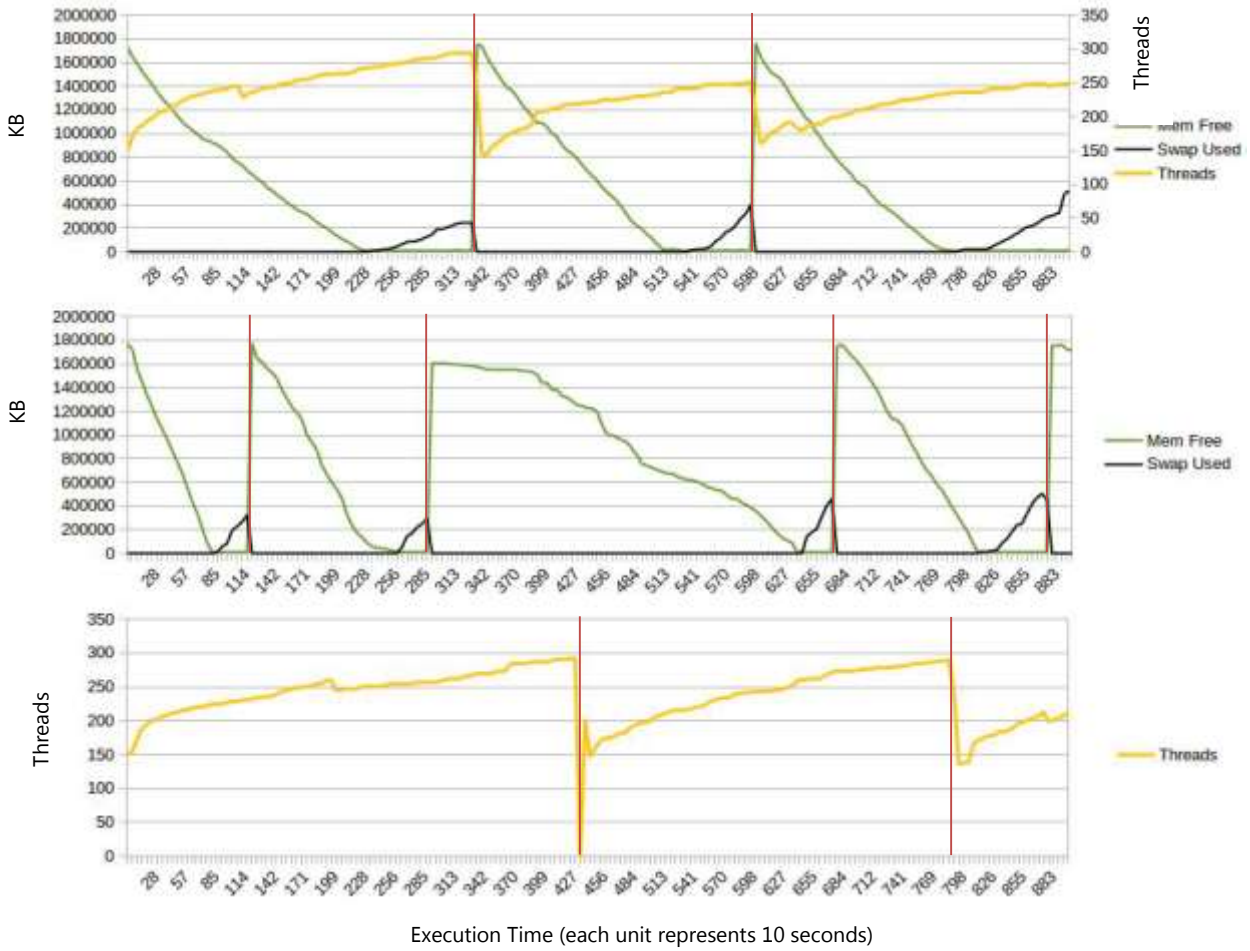

Figure 31: Main System Features for 3 couples of VMs

Figure 31 shows the interesting features of the 3 couples of VMs, along with the time instants where the Controller was switching from one VM to the other (the vertical red lines). The first plot reports the trend of parameters of the couple of VMs where both memory leaks and unterminated threads are inserted. The second plot is related to the couple of VMs where only memory leaks are injected. Finally, the third plot regards the couple of VMs where only unterminated threads are injected.

By the results, we can see that all the couples of VMs were suffering from the injected anomalies, and observe that the prediction model was actually able to identify the best place to rejuvenate one of the VMs of each couple, during the execution of the experiment. Of course, due to the stochastic nature of the injection of the anomalies, the red lines (thus the rejuvenation of one of the 2 VMs of the couple) appear at different points. Therefore, the controller is perfectly able to scale to a larger number of VMs (and of different prediction models, as well), without affecting the accuracy of the prediction.

7 CONCLUSIONS

In this report, we present the implementation of Virtualization and Machine Learning Frameworks for enhancing the availability and performance of web based applications. We automatically generated Machine Learning (ML) models, based on monitoring a set of system features, during the off-line training phase. Given the large number of system features to be monitored, we use Lasso regularization techniques for selecting a subset of system features, while preserving low values of prediction errors.

The implemented ML framework predicts the time to crash of applications, so that a proactive rejuvenation of the application can be periodically performed before the system fails and the response time is lower than a given threshold.

We created a working prototype of a system that allows self* properties (healing, rejuvenation, reconfiguring) and seamless application execution to be achieved. We realized a highly available web server by using a set of virtual machines.

This framework can be used for any cloud applications (not only web servers) that requires a large number of resources and has a high probability to fail, and runtime constraints.

We proposed a global architecture, based on ML framework, for a proactive management of cloud resources.

The developed architecture can be used for forming Hybrid Clouds and controlled by the Intra-Autonomic Cloud Managers (Intra-ACM). They are organised by the Inter-Autonomic Cloud Manager (Inter-ACM). It is in charge of orchestrating the Hybrid Clouds and the Private Local Virtual Infrastructures.

8 REFERENCES

[1] C. Cortes and V. Vapunik, "Support-Vector Networks," in *Machine Learning*, Kluwer Academic Publishers-Plenum Publishers, 1995.

[2] W. Ian H. and F. Eibe, Data Mining: Practical Machine Learning Tools and Techniques, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 2005.

[3] T. R., "Regression Shrinkage and Selection via the Lasso," *Journal of the Royal Statistical Society,* pp. 267-288, 1994.

[4] Z. Zheng and M. R. Lyu, "Selecting an Optimal Fault Tolerance Strategy for Reliable Service-Oriented Systems with Local and Global Constraints," *IEEE Transactions on Computers,* 2013.

[5] CloudWeaver, [Online]. Available: http://www.cloudweaver.com.

[6] E. Alpaydin, Introduction to Machine Learning, MIT Press, 2014.

[7] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, MIT Press, 1998.

[8] Y. Kodratoff and . S. Michalski, Machine Learning: An Artificial Intelligence Approach, Morgan Kaufmann Publishers, Inc., 2014.

[9] G. Hinton and t. J. Sejnowski, Unsupervised Learning, MIT Press, 1999.

[10] T. a. X. H. Adviser-Gu, "Online performance anomaly prediction and prevention for complex distributed systems," North Carolina State University, 2012.

[11] D. J. Dean and X. G. Hiep Nguyen, "Ubl: unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems," in *Proceedings of the 9th international conference on Autonomic computing*, 2012.

[12] C. Wang, V. Talwar, K. Schwan and P. Ranganathan, "Online detection of utility cloud anomalies using metric distributions," in *Network Operations and Management Symposium*, 2010.

[13] B. Sharma, P. Jayachandran, A. Verma and C. R. Das, "CloudPD: Problem determination and diagnosis in shared dynamic clouds," in *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010.

[14] PANACEA Project, "Deliverable 3.1: Implementation of a Virtualization framework at a node level of the overlay, based on open source software and developed in the project tools, for realizing the Machine Learning Framework," 2014.

[15] I. H. Witten, E. Frank, Hall and M. A., Data Mining: Practical Machine Learning Tools and Techniques, Third Edition, Morgan Kaufmann Series in Data Management Systems, 2011.

[16] RapidMiner, [Online]. Available: https://rapidminer.com/.

[17] MOA, [Online]. Available: http://moa.cms.waikato.ac.nz/.

[18] gnuplot, [Online]. Available: http://www.gnuplot.info/.

[19] MathWorks, "MATLAB - The Language of Technical Computing," [Online]. Available: http://www.mathworks.com/products/matlab/.

[20] LaTeX, [Online]. Available: http://www.latex-project.org/.

[21] 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 *Computer Architecture Evaluation Using Commercial Workloads*, 2000.

[22] Apache Foundation, "Apache Tomcat," [Online]. Available: http://tomcat.apache.org/.

[23] Oracle Corporation, "MySQL," [Online]. Available: http://www.mysql.com/.

[24] IBM, "An architectural blueprint for autonomic computing," 2005.