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

PANACEA

Proactive Autonomic Management of Cloud Resources

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

Version: v.1.0

Work package	WP 3
Task	Task 3.1
Due date	10/01/2014
Submission date	10/01/2014
Deliverable lead	IRIANCE (Leader-implemented sections 1 through 12), ATOS (prepared the appendix)
Version	1.0
Authors	Dimiter R. Avresky, David G. Perez
Reviewers	Ana Juan Ferrer (ATOS), Olivier Brun (CNRS)

Abstract	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.
Keywords	Machine Learning, prediction, seamless, availability, rejuvenation

Document Revision History

Version	Date	Description of change	List of contributor(s)
V1.0	10.01.2014	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

The implemented ML framework predicts the time to crash of applications, so that a proactive rejuvenation of the application(s) 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 reliable web server by using a set of virtual machines.

The Virtualization and Machine Learning Frameworks can be applied for any cloud applications (not only web servers) that requires a large number of resources and has a high probability to fail during the run time.

The Virtualization Framework allows to create scalable Private Clouds and Federation of Clouds.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

ML	Machine Learning
VM	Virtual Machine
WP	Work Package
M5P	Rational Reconstruction of M5
SVM	Support-Vector Machine
SVM2	Support-Vector Machine to the power of 2
RTTF	Remaining Time to Failure
RTTC	Remaining Time to Crash

1 INTRODUCTION

Availability is a critical aspects for many of computing systems. While in the past a large effort has been made by the research community to increase availability of single computing machines (e.g., by relying on some sort of hardware redundancy), in the last decades the focus has moved much more on distributed systems and software [1].

Two main approaches can be carried out to increase availability of distributed systems. On the one hand, we can try to devise an architecture which tries to delay the failure as much as possible (e.g., by relying on different instances of the techniques borrowed from older hardware redundancy). On the other hand, we can face the fact that the system will eventually crash, and perform specific operations aimed at minimizing the impact of the crash.

One significant measure used to evaluate the behaviour of a system is the Mean Time To Failures (MTTF). By relying on this measure, no difference is made on the cause of the failure, being it due to a hardware or software fault (yet, the effect is the same: the system could undergo a failure). In this project, we explicitly tackle software faults (with a focus on web servers), in order to devise an architectural framework which is able to proactively recognize the occurrence of problems related to the software layers (e.g., memory leaks, unterminated threads, ⋯), so as to undergo an early system rejuvenation which will in turn increase the overall system availability. This approach allows us to perform "conducted experiments" and to reduce the training period for validating the Virtualization and Machine Learning Frameworks.

We will explicitly rely on Machine Learning (ML) algorithms (namely, Support-Vector Networks [2], [4], [5], 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.

The ML framework, will be developed to monitor the utilization of hardware resources and create Knowledge Sources for predicting the time to crash of applications, and response time of web servers. The developed ML framework will demonstrate to be successful in the process of feature selection by reducing the number of parameters to a small and, in the meantime, providing a low value of Mean Absolute Error of the prediction, for Internet applications running in a quite dynamic environment and heavy workloads.

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

Overall, we will create a prototype of a system that is capable of autonomously healing itself. We will explicitly realize a highly reliable web server implemented within a set of virtual machines. The initial prototype will consist of three virtual machines VM1, VM2, and VM3, having VM2 and VM3 implement copies of the web server. VM1 will be the manager machine so that, whenever it detects that VM2 is about to fail, VM3 will be ready to step in and let VM2 resolve its problems. This scenario will be thoroughly discussed in Sections 4, 5, 6, and 10, and where we give as well a graphical representation (see Figure 5, Figure 20, Figure 31,

Figure 32).

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, and has high probability of failure. We emphasize that, although in the next part of the experimentation our initial prototype will consist of only 3 VMs, the system will be able to scale to any size, as we will show in the subsequent parts of the experimentation.

The document is organized in the following 9 sections: ML-Based Prediction Framework; Machine learning framework description; Experimental Environment and Data Flow for Machine Learning Framework; Virtual Environment for Machine Learning Framework; TPC-W: Test-bed System Description and Considerations; Machine Learning Framework: Prediction with Memory Leaks and 64 TPC-W Clients; Machine Learning Framework: Prediction with Memory Leaks, Unterminated Threads, and 64 TPC-W Clients; Discussion on the Learned Models; Dynamic Reconfiguration Based on ML Prediction.

2 ML-BASED PREDICTION FRAMEWORK

2.1 Motivation and State of the Art

One of the causes of applications/systems' hang or crash is the accumulation of errors, which are causing resources contention. These problems are particularly strengthened in the case of long running applications (such as web applications). Gradual performance degradation is typically observed as a consequence of accumulation of errors. These kind of error are often related to memory bloating/leaks, unterminated threads, data corruption, unreleased file-locks, overruns, etc. These phenomena have been widely observed in different contexts where many system crashes are caused by resource exhaustion.

An approach for coping with the afore-mentioned problems is based on continuous monitoring of system parameters, such as CPU utilization, memory usage, swap space, number of processes, workload, etc., in order to identify (and then to predict) possible side effects due to anomalies. For example, the side-effect of a memory leak is decreasing the free memory over time, which can cause a system crash. It is worth noting that, in order to make reliable predictions, a large number of variables should be monitored. However, monitoring so many variables can introduce a significant noise, which can affect the accuracy of the predictions. Additionally, the resource consumption over time could be non-linear. Hence, predicting the time to failure due to resource exhaustion is generally not a trivial task. A large variety of books and papers addressing analysis and prediction of failures in internet services have recently been published in the literature. Many publications focus on the use of virtualization for improving availability of internet services via software rejuvenation.

The prediction of failures in computer systems and Internet applications is quite challenging task. The following papers are presenting techniques for solving this problems: [7], [8], [10], [11], [13], [14] and [15]. Software rejuvenation is used for enhancement of availability and results are presented in [6], [15]. The anomalies and workloads can have significant impact on the software reliability and performance of distributed systems [12].

2.2 ML-Based Prediction Framework

As we mentioned, the ultimate goal of this research is to create an innovative framework for a proactive failure handling during the system' s operation. The framework leverages the concept of software rejuvenation for preventing possible system failures. Specifically, once having an estimation of the Remaining Time to Failure (RTTF) of a component of the system under prediction mode, a proactive action for rejuvenating the system (e.g. via early restart of a machine) is performed before the time the component is expected to fail. The time, when an action has to be performed, is established in order to reduce, as much as possible, the system unavailability. RTTF is predicted through a prediction model, which takes as input a set of variables which are monitored at runtime and related to utilization of system resources (e.g. CPU and memory etc..)

Given the high complexity of relations which may exist among utilization of system resources and the time to failure of the system, a ML-based approach will be used to build the RTTF prediction model. Specifically, our solution relies on off-line learning, where a number of data

samples, which are collected by observing resource utilization while the system running, are is used to build the prediction model through ML techniques. The model is then used to predict the RTTF on the basis of current measurements of the system resource utilization. On the basis of predicted RTTF, the rejuvenation mechanisms are activated.

As we mentioned before, one issue to deal with in this approach is that the number of involved variables can be high. Particularly, this factor can affect both the training time (which could become very long) and the created overhead, which is associated to the measurements to be performed. To cope with this issue, we utilize regularization techniques for reducing the number of parameters, which are exploited for on-line monitoring the system' s behaviour.

2.3 Impact of the Utilization of ML Framework for predicting anomalies

To further explain what are the implications of using the ML techniques for predicting anomalies, we will give an example. In Figure 1, we show how the whole system (comprising two Virtual Machines (VM)) would work without (a) and with (b) failure prediction.

Occurrence of a failure (F) determines how much has to be recomputed (dark-grey interval), as shown in Figure 1. After a failure occurs, the substitution unit has to be initialized (light-grey interval). After repair, the unit is ready and starts to redo the lost computations. When it has finished, the system is up again. Figure 1 shows two effects how failure prediction can reduce Time to Repair (TTR): The checkpoint (CP) may be established closer to the failure, and the substituting unit can be initialized, based on the ML based Proactive Rejuvenation framework, even before the failure occurs such that it is ready earlier after the failure.

The goal of these subtasks is to compute the impact of proactive fault handling on the system availability. Preliminary results show that the system availability can be increased i.e. for example **from 0.9999 to 0.99999**.

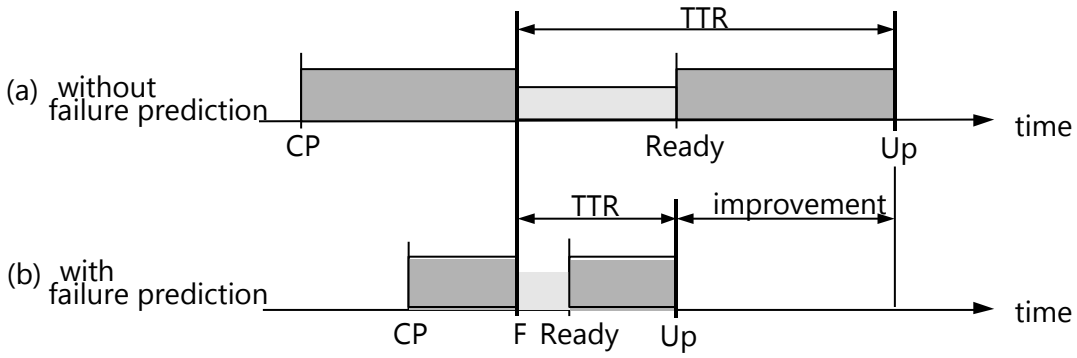

Figure 1: Impact of Prompt Rejuvenation

2.4 Availability Enhancement by Proactive Virtual Machine Rejuvenation

As mentioned, the Machine Learning Framework allows to select critical parameters for predicting anomalies. At this point, the infrastructure for generating the optimal training sets for the ML has been developed and tested completely. It can be applied for web servers,

cloud applications and any number of backup virtual machines can be created. It has ability to predict the RTTF with the use of machine learning techniques. The proposed new method, based on ML techniques, can be applied for increasing availability of web servers, cloud computing, software applications – computational intensive or service oriented (SaaS). Another opportunity is to learn to defend against more than one type of anomalies. In this case, whenever any anomaly occurs, proactive management of cloud recourses will be initiated.

The developed ML framework can be instantiated on each server to create the decision rules for its proactive rejuvenation for eliminating accumulated memory leaks and unterminated threads. The so-far obtained results (analytical and experimental) confirm that the proposed framework can reduce servers' unavailability.

Failure Prediction can be seen as the composition of two different actions. These actions can be driven by the failure probability based on system state and a data-driven approach.

System parameters such as memory usage, number of processes, workload, etc., can be monitored in order to identify side-effects of the faults. These side-effects are called syndromes. For example, the side-effect of a memory leak is that the amount of free memory decreases over time.

The goal is to compute the impact of proactive fault handling on the system availability, during the design time or system operations. We provide an equation that enables to calculate availability enhancement based on MTTF and MTTR from a system without proactive failure prediction.

Normally, availability A is determined by measuring and estimating MTTF and MTTR:

$$\mathrm{A} = \frac{\mathrm{MTTF}}{(\mathrm{MTTF} + \mathrm{MTTR})} \tag{1}$$

We introduce the *gains formula*, expressed as [14]:

$$\mathrm{A_{pfh}} = \mathrm{A_{orig}} + \mathrm{k} * \frac{\mathrm{MTTR}}{\mathrm{MTTF} + \mathrm{MTTR}} \tag{2}$$

where, $\mathrm{k} = 1 - \left(1 + \mathrm{r} * (\mathrm{rf} - 1)\right) * \left(1 - \mathrm{P_p} * \mathrm{r} + \mathrm{P_e} * \frac{\mathrm{r}}{\mathrm{p}}\right)$, and:

- $\mathrm{A_{orig}}$ is the availability of the system without proactive failure handling;
- $\mathrm{A_{pfh}}$ is the availability with proactive failure handling.

k depends on:

- Precision (# correct alarms/all alarms): p
- Recall (# correct alarms/ all failures): r
- Probability that a failure can be avoided: $\mathrm{P_p}$ (failure prevention)
- Probability that failures occur due to actions: $\mathrm{P_e}$ (extra failure)
- Mean improvement in MTTR: rf

An Example

If we assume a 4-Nines System, we have that:

Correctness of Failure Prediction:

- Alarms are correct in 90% of all cases (precision), (p) ;
- 90% of all failures are predicted (recall), (r);

Performance of the methods:

- Correct alarms: Prevention probability = 95% (P_p)
- Correct alarms: Improved repair time = 0.5 * MTTR
- False alarms: Probability of extra failures = 15% (P_e)

Availability of original system:

- A_{orig} = 0.999900

Availability with a proactive failure handling:

- A_{pfh} = 0.999983

It is worth noting for the range of the values of the parameters that have been used for accessing the quality of a Proactive Failure Handling (p>90%, r>90%, P_p>95%, rf<0.5, and P_e<15%), the developed Proactive VM-framework with Machine Learning (ML) techniques can provide a higher availability than original (A_{orig} = 0.9999, A_{pfh} = 0.999983), i.e. can significantly improve the availability results with proactive failure handling.

The observed parameters will be reduced by the Proactive VM-rejuvenation framework with Machine Learning techniques and utilized for predicting the time to crash of computing system due to the software ageing. The experimental results will demonstrate that the implemented framework is capable of predicting accurately the time to crash of the system due to memory leaks and threads.

3 MACHINE LEARNING FRAMEWORK DESCRIPTION

The Machine Learning Framework is based on a set of utility, which are governed by a Bash shell script. The components are `aggregate.c`, `lasso.m`, `applyBeta.py`, *WEKA*, `plot.plt` and `evaluateErrors.py`.

To control the workflow, the 000.do_run.sh script can be used. Specifically, at the beginning of the script the following variables can be set:

```
# Framework Configuration
database_file=aggregated-db.txt
lambdas=(0.1  1  10  100  1000  10000  100000  1000000 10000000 100000000 1000000000)
WEKA_PATH="c:/Program Files/Weka-3-7/weka.jar"

# What do we have to run?
run_datapoint_aggregation=true
run_lasso=true
apply_beta_vector=true
plot_original_parameters=true
plot_lasso_as_predictor=true
run_weka_linear=true
run_weka_m5p=true
run_weka_svm=true
run_weka_svm2=true
evaluate_error=true
```

The first variables specify which is the input database file (generated by the feature monitor client/server architecture) to generate `aggregated.csv`, the lambda values for which to run Lasso, and the path to WEKA on the current machine.

Then, the set of flags state which portions of the framework must be activated when running `./000.do_run.sh`

3.1 Feature Selection with Lasso

The script `lasso.m` performs feature selection based on *lasso regularization*. For each element λ of vector *lambdas*, the script `lasso.m` is executed and provides as output the vector β, whose elements are the *weights of the* vector $\mathbf{x_j}$ which minimizes the following objective function:

$$\frac{1}{n}\sum_{j=1}^{n} V\left(y_j, \langle \beta, x_j \rangle\right) + \lambda \|\beta\|_1 \qquad (3)$$

where $\mathbf{n}$ is the number of data points included in the output of the script *aggregate.py*, $\mathbf{x_j}$ is a vector of values of input features (independent variables) of each data point, $\mathbf{y_j}$ is the associated value of the dependent variable (remaining time to failure of the system) for the specific data point, and $\mathbf{V}(\mathbf{y_j}, \langle \boldsymbol{\beta}, \mathbf{x_j} \rangle)$ is equal to $\left(\mathbf{y_j} - \boldsymbol{\beta}^{\mathbf{T}}\mathbf{x_j}\right)^{\mathbf{2}}$.

For each value of λ, 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. 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.

3.2 Aggregation of Datapoints

The `aggregate` program is used to generate aggregated datapoints on the basis of a threshold (in seconds) given at compile time. The default value is 30 seconds. The additional goal of this program is to add additional features to the aggregated datapoints, namely slopes. Each input datapoint has the following structure:

```
Datapoint: 241.498324
Memory: 515580 397752 117828 0 5976 72168
Swap: 409616 174980 234636
CPU: 23.650000 0.010000 1.280000 0.040000 0.000000 75.029999
```

And the meaning of each number on each line is the following:

- Datapoint:
  - Generation time: it is the time on the VM under control at which the datapoint has been generated. Time 0 is considered to be the initial time at which the VM became operational.
- Memory:
  - Total: the amounts (in bytes) of total available memory on the VM;
  - Used: the amount (in bytes) of the memory used by any application running on the VM at the time the specific datapoint was generated;
  - Free: the amount (in bytes) of unused memory on the VM at the time the specific datapoint was generated;
  - Shared: the amount (in bytes) of memory used as shared buffers among different applications on the VM at the time the specific datapoint was generated;
  - Buffers: amount of memory used for OS buffer cache
  - Cached: data which is no longer used by application, but is left in main memory to enhance spatial/temporal data locality.
- Swap:
  - Total: the amounts (in bytes) of total available swap space on the VM;
  - Used: the amount (in bytes) of the swap space used on the VM at the time the specific datapoint was generated;
  - Free: the amount (in bytes) of unused swap space on the VM at the time the specific datapoint was generated;
- CPU:
  - User: percentage of CPU time spent in user mode;
  - Nice: percentage of CPU time spent in user mode for processes with lower nice value;
  - System: percentage of CPU time spent in system mode;
  - I/O Wait: percentage of CPU time spent waiting for I/O;
  - Steal: percentage of time a virtual CPU waits for a real CPU while the hypervisor is servicing another virtual processor;
  - Idle: percentage of CPU time spent doing nothing useful.

The aggregation is done according to the scheme shown in Figure 2.

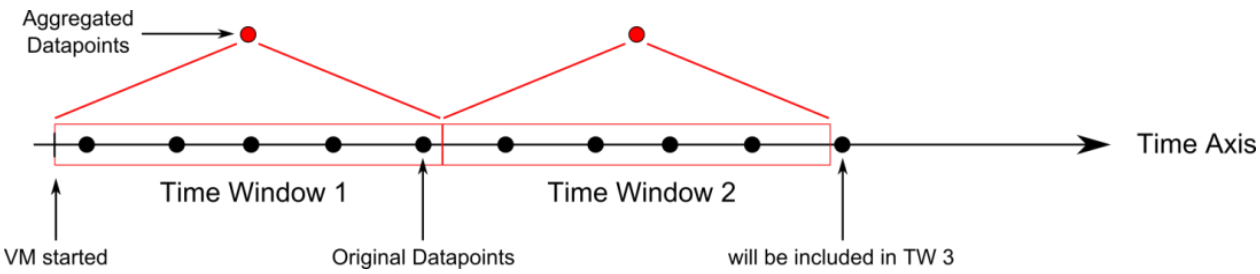

Figure 2: Datapoint Aggregation Scheme

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.

Then, for each of the original measurements, slopes are computed according to the following formula:

$$slope = \frac{start - end}{n} \tag{4}$$

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

Then, the time window is moved forward, so that in general any time window starting point can be identified as:

$$T_i = T_{i-1} + threshold \tag{5}$$

And so a new aggregated datapoint is generated. This behavior allows us to compute the measurements for each aggregated datapoint using, on the one hand, a variable number of datapoints, but gives of, on the other hand, a more precise photograph of the system in a given time window, due to the fact that the generation of original datapoints might incur in some skewing do to, e.g., the scheduler of the operating system, depending on the current workload. Therefore, as soon as we approach the crashing point, it is expected that the impact this skew becomes more prominent, and therefore not taking into account it might result in incorrect forecasting on the VM Rejuvenation side.

3.3 Machine Learning Framework

Figure 3 shows all steps of the ML Framework from system monitoring to prediction model building.

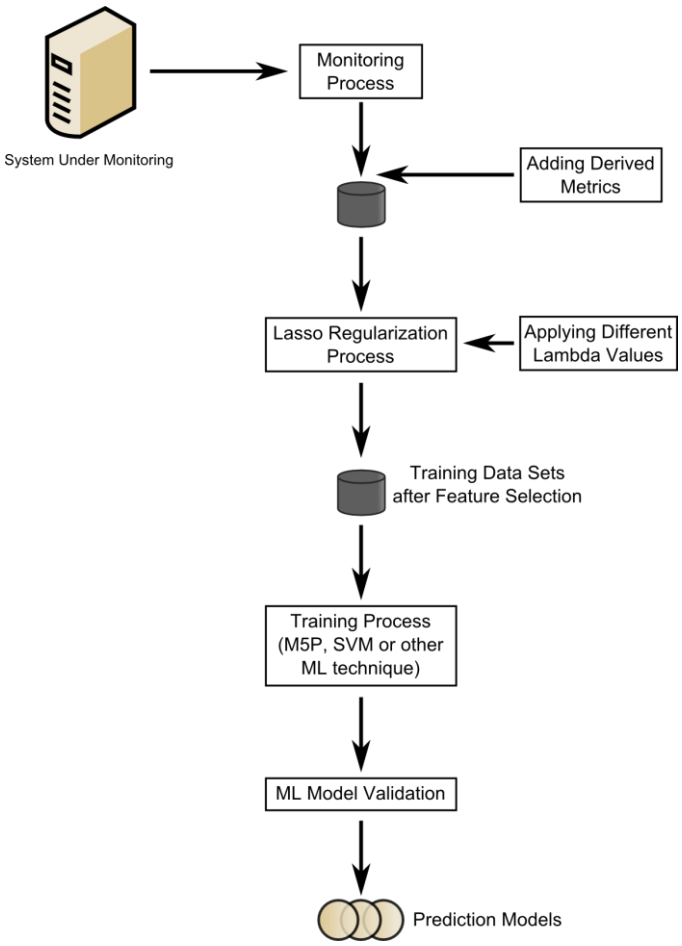

Figure 3: Machine Learning Framework

3.4 Injection of Anomalies

Two different anomalies are injected in the TPC-W server, namely *memory leaks* and *unterminated threads*. As we will show, they have different impacts on the overall execution pattern of the system. To correctly mimic the behaviour of a system affected by software errors, anomalies are injected according to statistical distributions.

3.4.1 Memory leaks

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 behavior 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.

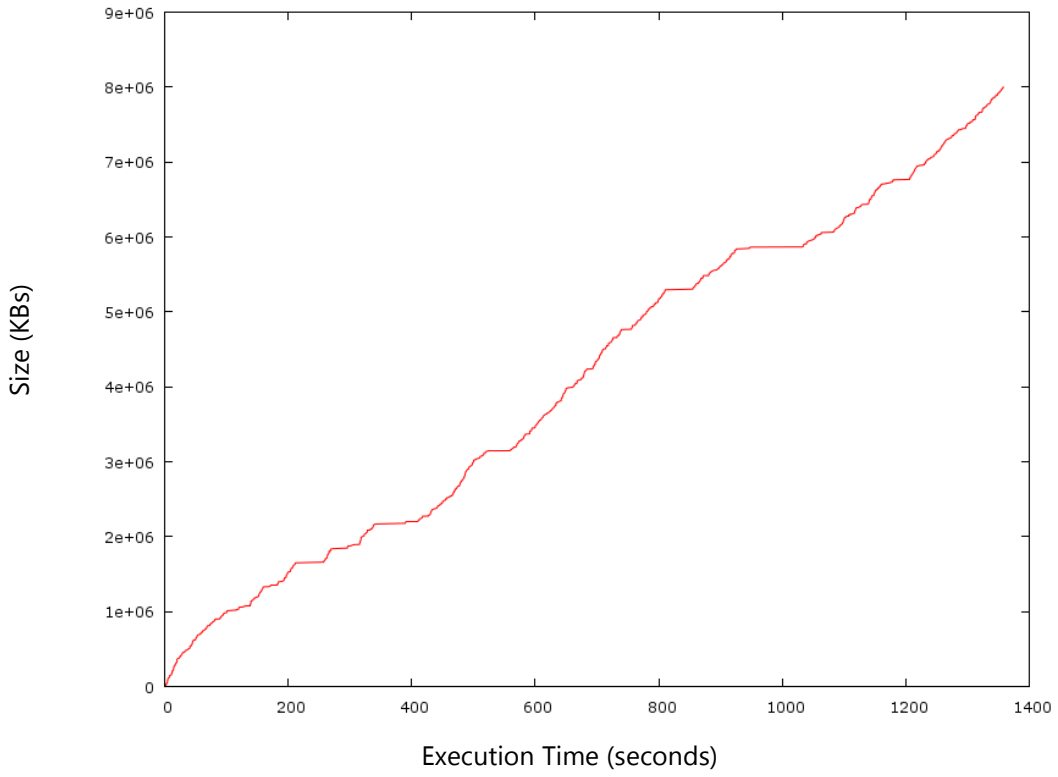

Figure 4: Memory leak injection

3.4.2 Unterminated Threads

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.

4 EXPERIMENTAL ENVIRONMENT AND DATA FLOW FOR MACHINE LEARNING FRAMEWORK

The experimental environment architecture and the data flows are depicted Figure 5.

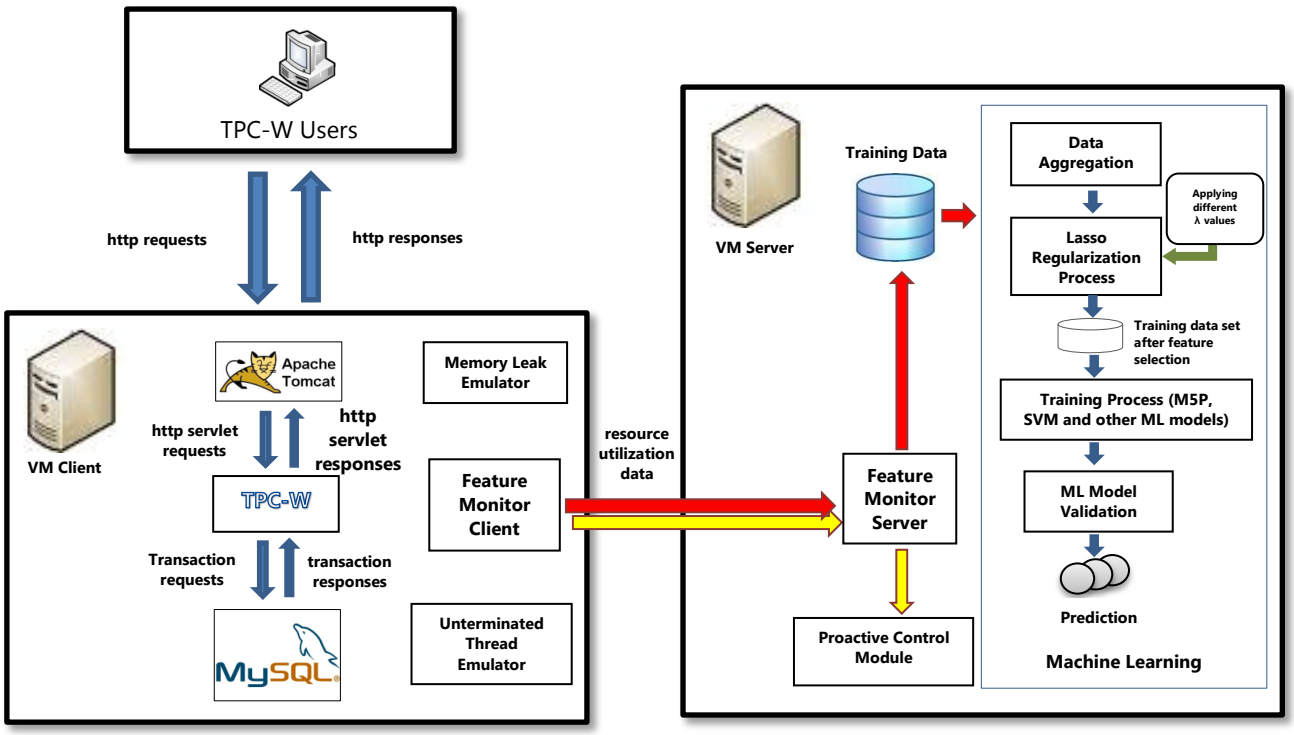

Figure 5: Experimental virtual environment and data flow for building the ML Framework

VM client can be consider as a Server to the TPC-W users and as a client to VM Server.

4.1 Description of Software Components

The software components of the experimental environment are: TPC-W User, VM Client.

Each VM Client is equipped with:

- Apache Tomcat 6.0 (6.0.41)
- A Java implementation of TPC-W (The Java TPC-W Distribution version 1.0 release by University of Wisconsin-Madison, http://pharm.ece.wisc.edu/tpcw.shtml)
- MySql Server version: 5.1.41-3
- A *Memory Leak Emulator*, which is a proprietary software package implementing a generator of memory leaks

- An *Unterminated Threads Emulator*, which is a proprietary software package implementing a generator of unterminated threads
- A *Feature Monitor Client* , which is a proprietary software package aimed to collect 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.

VM Server

- A *Feature Monitor Server*
- The *Machine Learning Framework*

4.2 Main Configuration Parameters

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).

The memory leak injection configuration parameters are: the injection rate (injected memory leaks per second, drawn from an exponential distribution); the size of memory leaks to be injected (minimum and maximum value of the uniform distribution).

The unterminated threads' configuration parameters are: the spawning rate (the average of the exponential distribution is drawn uniformly at random, at each injection step from the given interval of seconds).

4.3 Data Flows

The system is featured by independent data flows:

1) **TPC-W data flow (blue arrows in Figure 5.)**
   1.a) The client sends a http request to Apache Tomcat
   1.b) Apache Tomcat sends a http servlet request to the TPC-W
   1.c) TPC-W sends a query(*)/transaction(**) request to MySql Server
   1.d) MySql Server sends a transaction/query response to TPC-W
   1.e) TPC-W sends a http servlet response to Apache Tomcat
   1.f) Apache Tomcat sends a http response to the client

   (*) a query is a read-only request for a set of data stored in the the database
   (**) a transaction is a read-write request for a set of data in the database

2) **Machine Learning Framework data flow (red arrows in Figure 5.)**
   2.a) The Feature Monitor Client collects resource utilization data of the VM client and sends them to the Feature Monitor Server

2.b) The Feature Monitor Server stores resource utilization data in the Training Data Store

2.c) The Machine Learning Framework gets data from the Training Data Store for building prediction models.

3) **Prediction data flow (yellow arrows in Figure 5)**

3.a) The Feature Monitor Client collects resource utilization data of the VM client and sends them to the Feature Monitor Server (same as 2.a)

3.b) Features are passed to the proactive control module for prompt failure prediction

4.4 Activation of the Machine Learning Framework

To effectively rely on the proposed ML Framework for VM Rejuvenation, an initial training is required. The software architecture depicted in Figure 5 is capable of both predicting forthcoming failures and learn the specific behavior of an application. In both cases, the key aspect are the Feature Monitoring client/server components.

When the system is started, the Feature Monitoring client periodically reads system parameters (memory usage, swap usage, CPU usage, ⋯) and transmits the data to the server, which is in charge of storing them in a database. When running in *training mode*, the server side of the framework can be (manually) triggered to learn the behavior of the system. In this case, the Machine Learning Framework (the behavior of which has already been described) is activated. The outcome of the ML Framework is a prediction function, which is automatically installed into the system at the end of the learning procedure.

On the other hand, if a prediction function has already been installed, the system can run as well in a proactive mode. This execution mode is different from the training one as the Feature data coming from the Feature Monitoring client are processed immediately. The system can be configured to decide whether the same data should be stored in the database as well or not. The former configuration could allow for a subsequent re-training of the system, relying on an increased amount of training data.

When in a proactive mode, the data coming from the Feature Monitoring client are immediately passed to the Decision Module. It evaluates the prediction function learned from the ML Framework, and if it detects that the currently running system is approaching a failure state, then it triggers VM Rejuvenation.

5 VIRTUAL ENVIRONMENT FOR MACHINE LEARNING FRAMEWORK

The experimental environment used for our experimentation (Figure 6) 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 10.0.3 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)

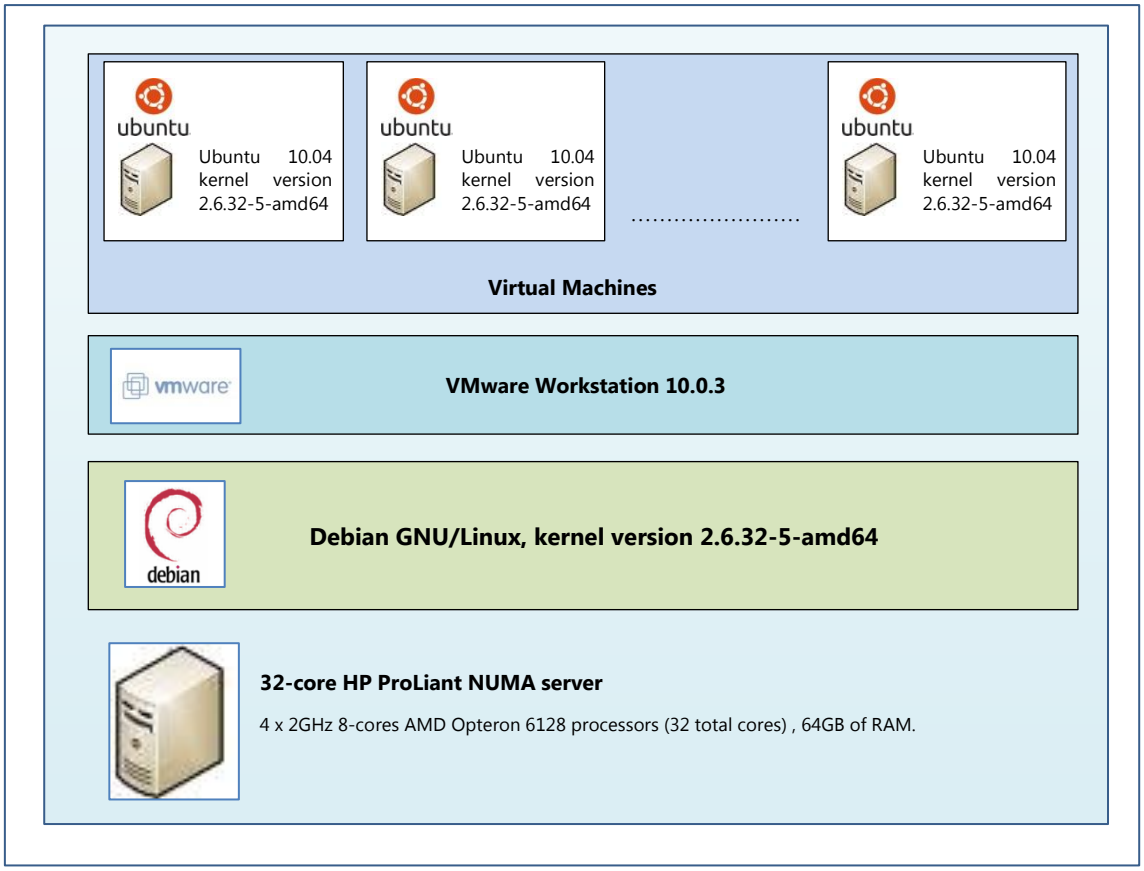

Figure 6: Experimental Virtual Environment

5.1 Virtual Network Topology

In Figure 7, 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 and resilient to partitioning in presence of multiple node failures.

In Figure 8, we show two virtual networks (each one including 8 nodes) hosted in different physical machines, which communicate through the Internet.

In our virtual framework we can install up to 30 virtual machines, each one with one virtual core and 2 GB of RAM. 2 cores and 4 GB of RAM are dedicated to the hosting operating system and the hypervisor).

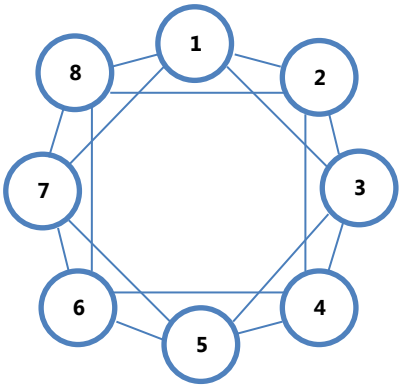

Figure 7: Virtual Network Topology with 8 Nodes

All the VMs are connected via standard TCP/IP sockets. The additional (physical) nodes added to the system can be either local, or geographically distributed, due to the standard nature of TCP/IP.

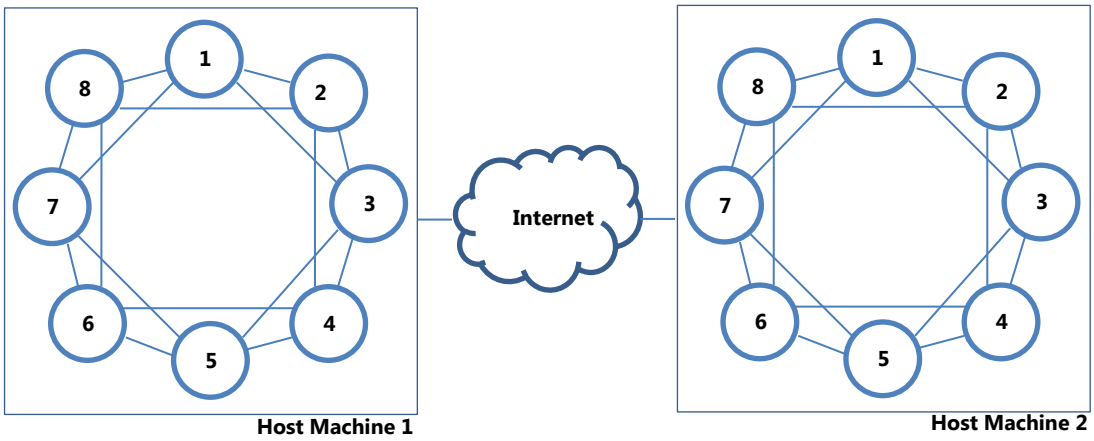

Figure 8: Two Virtual Network Topologies hosted in different Physical Host Machines over the Internet

The virtual topologies can be used for forming Private Clouds and controlled by Intra-Autonomic Cloud Managers (Intra-ACM). In this way, Inter-ACM can control the behavior of the Federated Clouds and can assure their autonomic properties.

6 TPC-W: TEST-BED SYSTEM DESCRIPTION AND CONSIDERATIONS

6.1 User Navigation Diagram

In Figure 9, the user navigation diagram is depicted. Each node in the graph represents a web interaction between the user and the system. Arrows represent the next interaction(s) performed by the user starting from a given state (web page). As an example, from the Home page, the user can decide to move to one of the following web pages: Search Request, Best Seller, New Product, Shopping Cart and Order Inquiry.

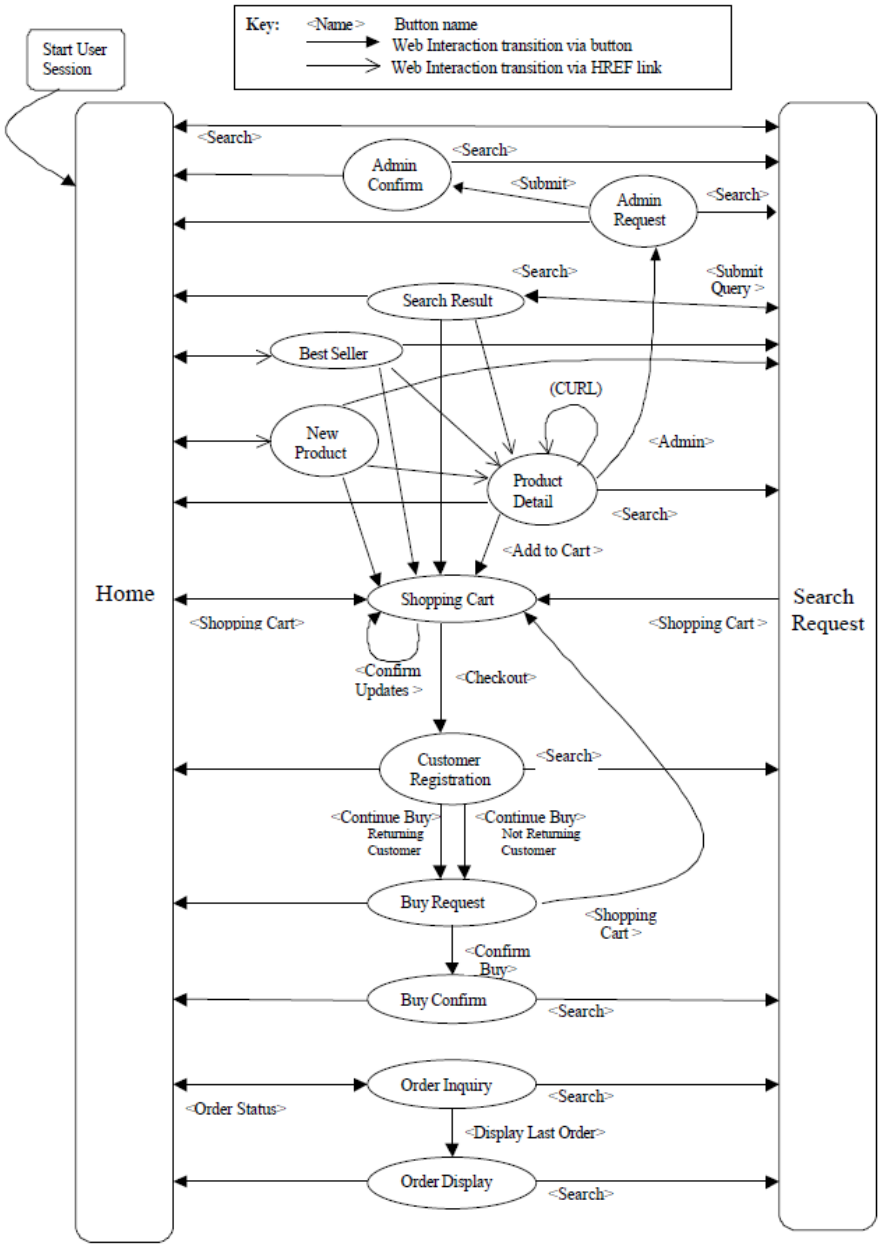

Figure 9: TPC-W user navigation diagram

To generate TPC-W requests, an emulated browser is used. This browser, implemented as a set of Java classes, issues requests to the TPC-W web servlet depending on a set of configuration parameters.

To define the class of web accesses issued by the emulated browser, a certain specific object should be instantiated at startup. The set of available classes are:

To generate TPC-W requests, an emulated browser is used. This browser, implemented as a set of Java classes, issues requests to the TPC-W web servlet depending on a set of configuration parameters.

To define the class of web accesses issued by the emulated browser, a certain specific object should be instantiated at startup. The set of available classes are:

- EBTPCW2Factory: Produces TPCW EBs for transistion subset 2 (The Shopping Mix - WIPS)
- EBTPCW1Factory: Produces TPCW EBs for transistion subset 1 (The Browsing Mix - WIPSb).
- EBTPCBFactory: Produces TPCW EBs for transistion subset 1.
- EBTPCW3Factory: Produces TPCW EBs for transistion subset 3 (The Ordering Mix - WIPSo).
- EBWWWFactory: Produces TPCW EBs for transistion subset 1.

Additionally, the emulated browser can be configured with a specific *think time*, which emulates the time spent by an user browsing the content of the web page provided by the client.

To carefully evaluate the Response Time of the web application, we have introduced software probes in the emulated browsers to store on a database file the response time of every web interaction which is carried out.

To carefully evaluate the Response Time of the web application, we have operated a slight change to the RBE.java class. Specifically, we have altered the instantiation of the statistics object, by changing the interval time according to which statistics are reported. This is reflected in the following line of code:

```
rbe.stats = new EBStats(rbe, 60000, 1, 75000, 100, ebfArg.maxState, start, 1000L*ru.num,
                                                    1000L*mi.num, 1000L*rd.num);
```

Where the value 1 allows to gather statistics on a 1 millisecond basis.

By this change, the Response Time of the application when varying the workload are reported in the following plots as Inverse Cumulative Distribution Functions (Inverse CDF), 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. Therefore, if on the x axis, at time 0.2 we find the value 10% on the y axis, it means that 10% of the web interactions take more than 0.2 seconds to complete.

6.2 Selection of the Best-Suited TPC-W Configuration

The following plots show the Inverse CDF of response time averaged over the various web interactions. It is a more meaningful measure for our study with respect to the TPC-W

standard (which shows the Inverse CDF for each type of interactions) because we are not interested in the response time of specific web interactions, rather we want to know how the whole system *on average* is behaving. This allows us to select the best configuration scenario for our experimentation where the workload is non minimal, yet is not showing thrashing phenomena, in normal conditions, *independently of* the specific types of involved web interactions.

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. Configurations with a higher number of clients (e.g., 128, as in Figure 13) show response times which can be higher than 0.5 seconds. These values are not plotted to allow for an easy comparison among the different configurations. In fact, if the green line converges quickly to zero, the system is not overloaded. In configuration where (in the given interval) the curve does not converge to zero, the system is sustaining a high workload (e.g., 64 clients, as in Figure 12) or is already thrashing (as in Figure 13).

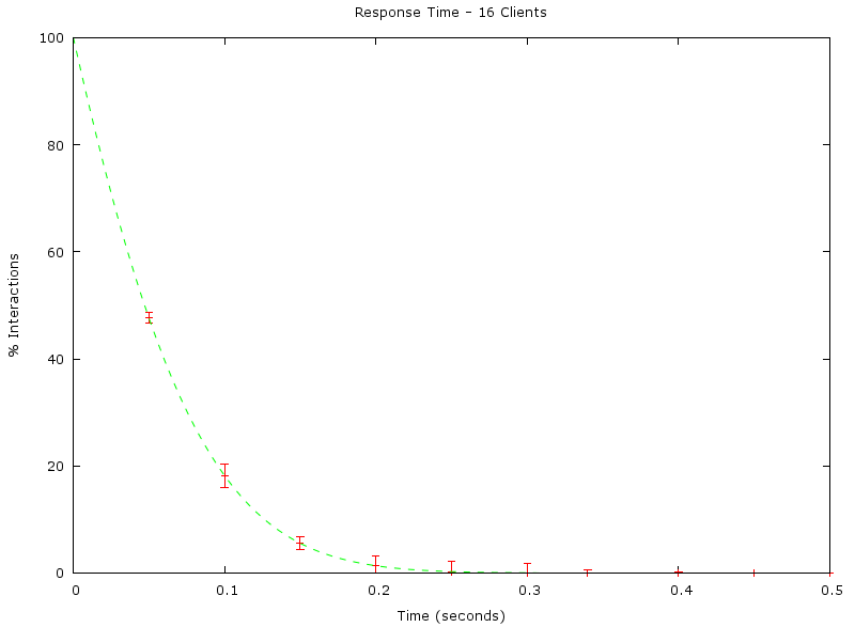

Figure 10: TPC-W Response time with 16 Users

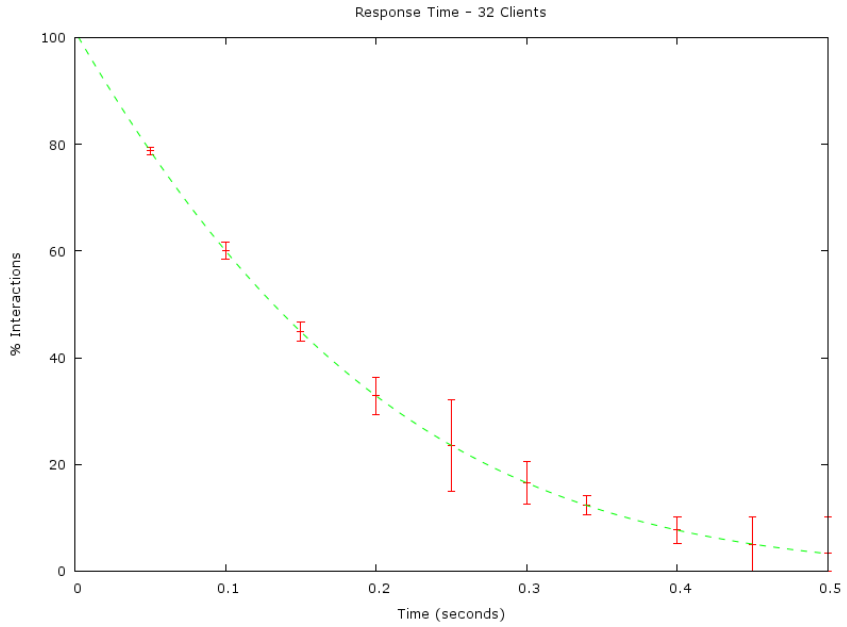

Figure 11: TPC-W Response time with 32 Users

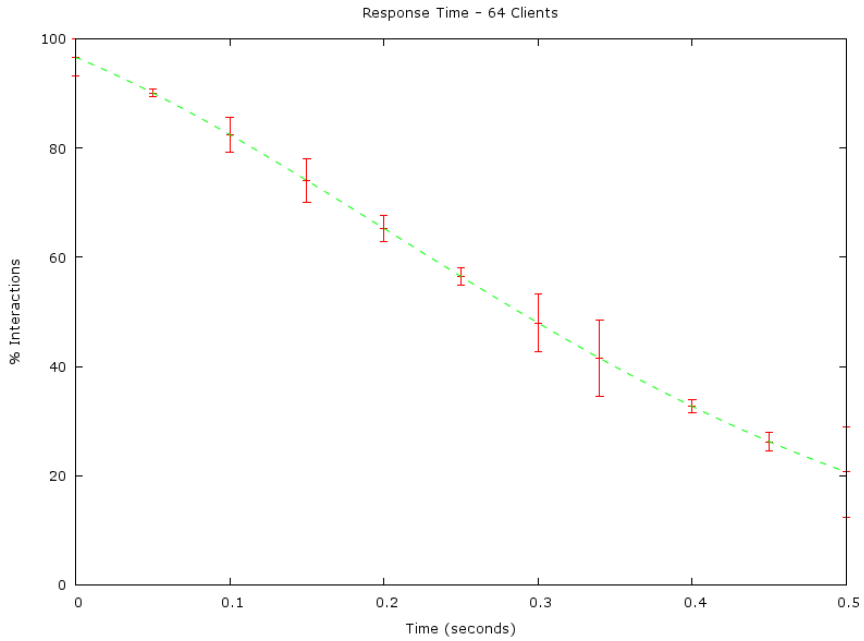

Figure 12: TPC-W Response time with 64 Users

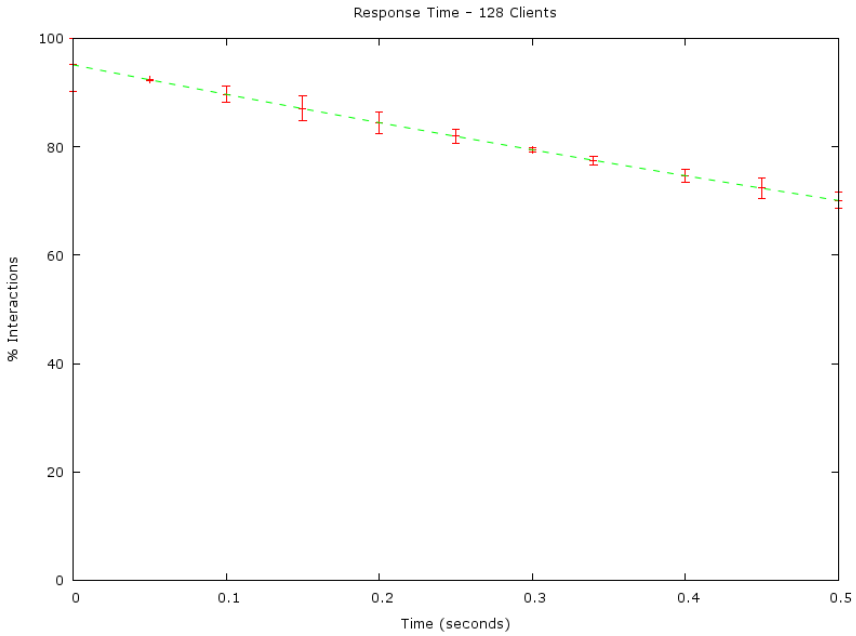

Figure 13: TPC-W Response time with 128 Users

By these results, considering the above discussion, we have decided to carry on the remainder of our experimentation when dealing with 64 clients. This choice 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.

7 MACHINE LEARNING FRAMEWORK: PREDICTION WITH MEMORY LEAKS AND 64 TPC-W CLIENTS

We have collected data related to the behaviour of the TPC-W server when injecting memory leaks up to the collection of **21 MiB of raw database points**. This data has been feed into the Machine Learning Framework to generated the aggregated database file as described before, and we report in this section the outcome of our learning framework.

7.1 Behaviour of the System

Figure 14 shows the system parameters (before feeding them into Lasso) on a large scale machine (HP ProLiant server), with an increased amount of computing power. The dotted curves (cpu usage) are to be read against the right-hand side y axis, while the solid ones (memory usage) are to be read against the left-hand size y axis.

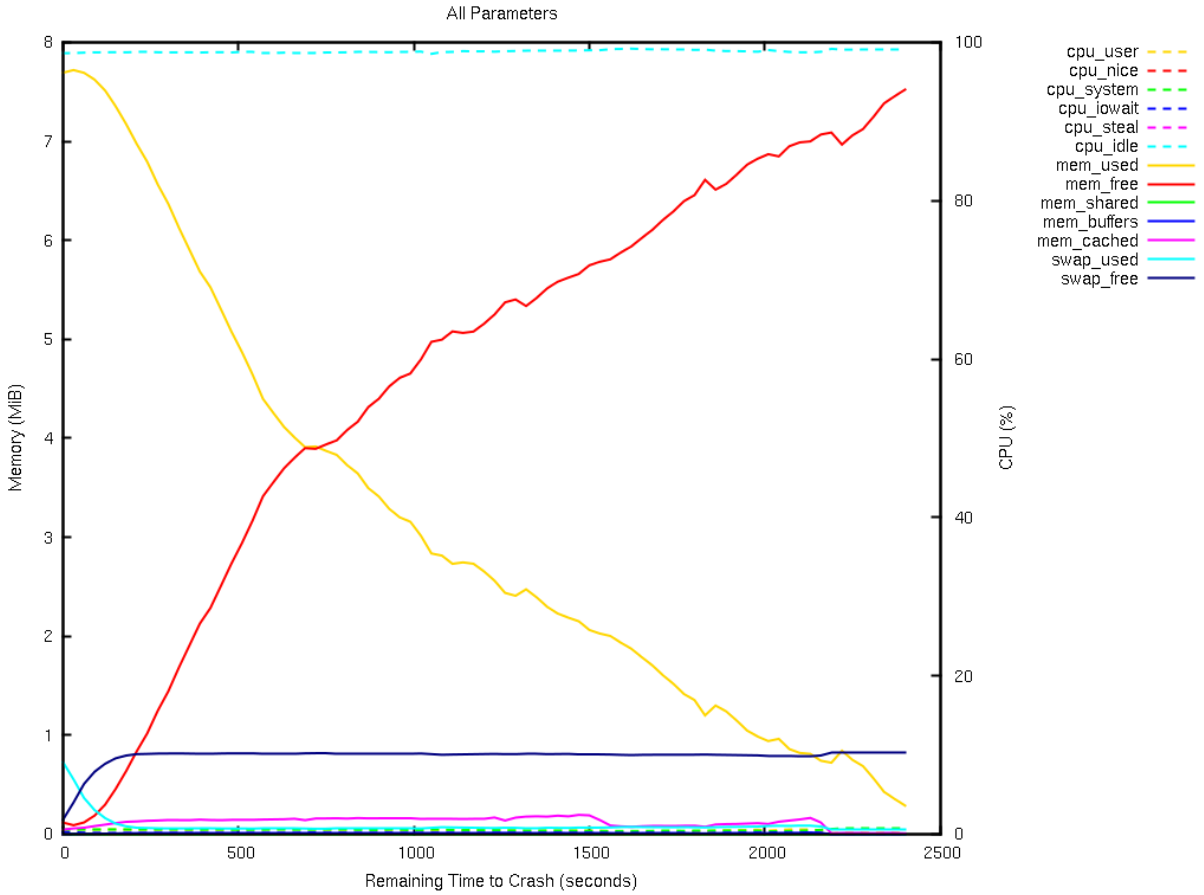

Figure 14: System parameters (memory leaks)

It is clear by the results that the HP ProLiant server provides, due to the increased power, more stable results which are affected only by memory leaks. In this scenario, Lasso will select less parameters even with small values of lambda, and the weights chosen by Lasso will be smaller.

7.2 TPC-W Response Time

In Figure 15, we report the Response Time of the TPC-W web application when injecting memory leaks in the system. Specifically, this is an average of the response time as seen by all the 64 clients.

Initially, the response time is very reduced (never higher than 0.5 seconds, usually around 0.2 seconds). When the system starts to suffer from memory leakage (around 1000 seconds) the response time increases, but stays nevertheless on a sustainable value (less than 1 second). When the system starts to swap data out of main memory (around 1200 seconds), the response time increases over an acceptable threshold, reaching a value of 2.5 seconds just before the system crash.

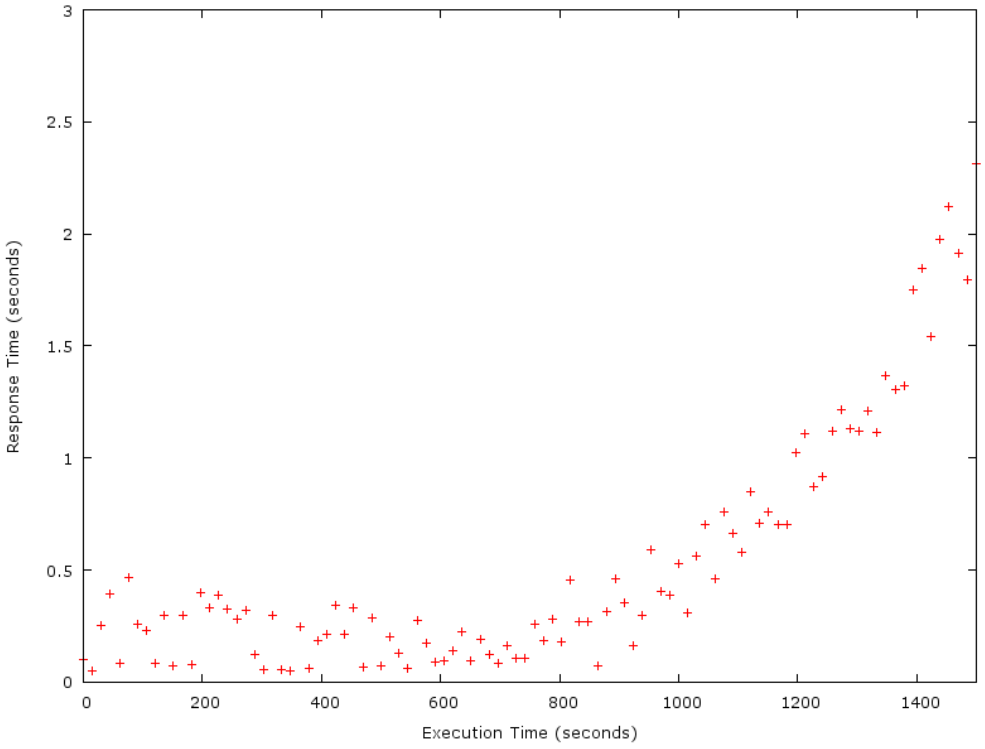

Figure 15: TPC-W (64 clients) Response time (memory leaks)

7.3 Parameters Selected by Lasso

The amount of selected parameters according to the different values of λ is reported in Table 1.

λ	# Parameters		λ	# Parameters
0.1	3		100000	3
1	3		1000000	3

10	3		10000000	4
100	3		100000000	4
1000	3		1000000000	4
10000	3			

Table 1: Parameters Selected by Lasso (memory leaks)

The exact parameters selected by Lasso when varying λ and the weights associated with them is shown in Table 2.

$\lambda = 0.1$	mem_used	0.000013649385818
	mem_free	0.000187583948520
	mem_cached	0.000184478736579
$\lambda = 1$	mem_used	0.000013649385818
	mem_free	0.000187583948520
	mem_cached	0.000184478736579
$\lambda = 10$	mem_used	0.000013649385818
	mem_free	0.000187583948520
	mem_cached	0.000184478736573
$\lambda = 100$	mem_used	0.000013649385819
	mem_free	0.000187583948520
	mem_cached	0.000184478736517
$\lambda = 1000$	mem_used	0.000013649385835
	mem_free	0.000187583948523
	mem_cached	0.000184478735955
$\lambda = 10000$	mem_used	0.000013649385988
	mem_free	0.000187583948550
	mem_cached	0.000184478730337
$\lambda = 10000$	mem_used	0.000013649387523
	mem_free	0.000187583948824
	mem_cached	0.000184478674154
$\lambda = 100000$	mem_used	0.000001697351691
	mem_free	0.000171061754878
	mem_cached	0.000167710328176
	swap_free	0.000141742532774

$\lambda = 1000000$	mem_used	0.000016615566599
	mem_free	0.000186954848952
	mem_cached	0.000170612459978
	swap_free	-0.000105062409883
$\lambda = 10000000$	mem_used	0.000016620485923
	mem_free	0.000186952156977
	mem_cached	0.000170541549926
	swap_used	-0.000105155487097
$\lambda = 100000000$	mem_used	0.000016768446011
	mem_free	0.000186780540399
	mem_cached	0.000170054815553
	swap_used	-0.000104409673668
$\lambda = 1000000000$	mem_used	0.000016768446011
	mem_free	0.000186780540399
	mem_cached	0.000170054815553
	swap_used	-0.000104409673668

Table 2: Weights assigned by Lasso, when injecting memory leaks

7.4 Trend of Parameters selected by Lasso

In Figure 16 we show variations of input features averaged for all executed runs, when using $\lambda = 10^9$, which provides minimum prediction error.

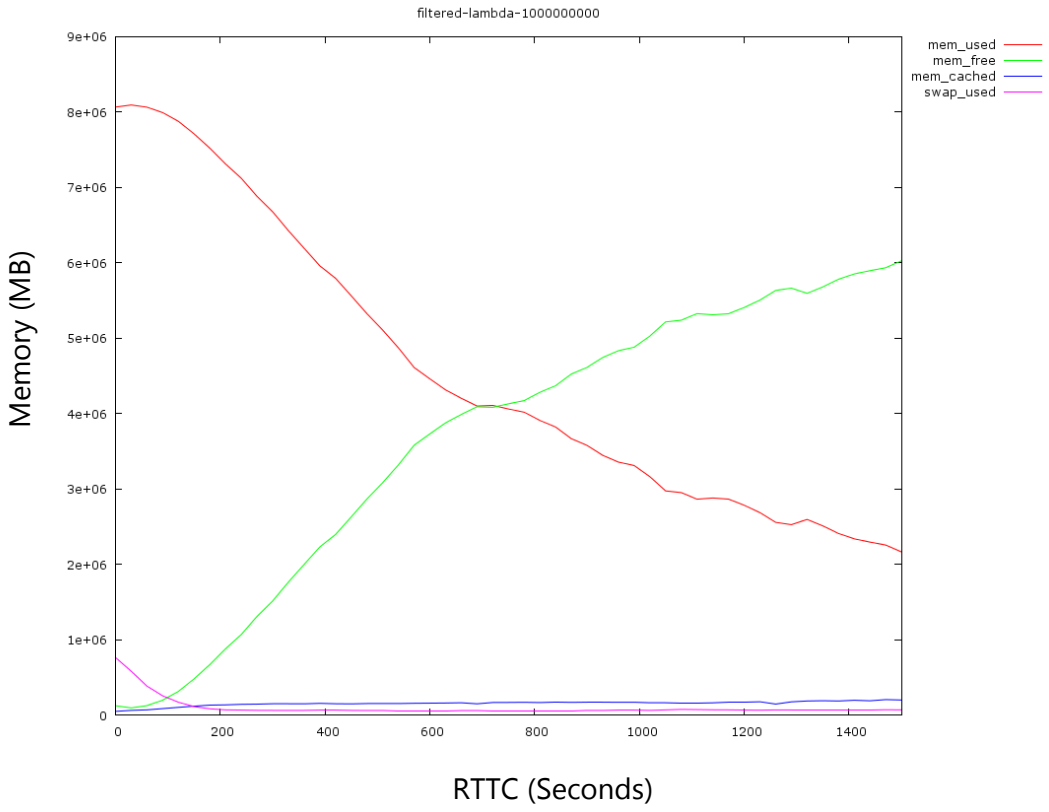

Figure 16: Parameters Selected by Lasso (memory leaks, $\lambda = 10^9$)

7.5 Machine Learning Model Building

Once calculated values of the vector β for each value of λ, WEKA is used for building machine learning models for predicting the Remaining Time To Failure (RTTF) of the system. We used three learning methods, i.e. Linear Regression, M5P (decision tree with the possibility of linear regression functions at the nodes) and Supported Vector Machine (SVM). Before building learning models, the script `applyBeta` is executed for producing input files to WEKA. The script filters out all features for which the calculated weight of the vector β is equal to zero. Thus, the remaining features are used for training the machine learning models through the three above-mentioned learning methods. In the following, we show of the results we achieved used the different methods.

7.5.1 Maximum Absolute Prediction Error

This error represents the highest error (worst case) encountered during the prediction. It is expressed in seconds, and the corresponding values are reported in Table 3.

Lambda Value	LASSO as a predictor	Linear Regression	M5P	SVM	SVM2
0.1	1229.958345	218.391	134.8262	217.0491	217.0500
1	1229.958345	218.391	134.8262	217.0491	217.0500
10	1229.958345	218.391	134.8262	217.0491	217.0500

100	1229.958345	218.391	134.8262	217.0491	217.0500
1000	1229.958345	218.391	134.8262	217.0491	217.0500
10000	1229.958345	218.391	134.8262	217.0491	217.0500
100000	1229.958344	218.391	134.8262	217.0491	217.0500
1000000	1202.564824	215.5886	106.1356	213.9006	213.9901
10000000	1208.739091	215.5886	106.1356	213.9006	213.9901
100000000	1208.704760	215.5886	106.1356	213.9006	213.9901
1000000000	1207.978073	215.5886	106.1356	213.9006	213.9901

Table 3: Maximum Absolute Prediction Error (memory leaks)

7.5.2 Relative Absolute Prediction Error

The relative absolute E_i 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.

$$E_i = \frac{\sum_{j=1}^{n}|P_j - T_j|}{\sum_{j=1}^{n}|T_j - \overline{T}|} \quad (6)$$

where P_j is the value predicted for sample j (out of n samples), $T_j$is the target (i.e., ground truth) value for sample j, and:

$$\overline{T} = \frac{1}{n}\sum_{j=1}^{n} T_j \quad (7)$$

It is expressed in percentage, and the corresponding values are reported in Table 4.

Lambda Value	LASSO as a predictor	Linear Regression	M5P	SVM	SVM2
0.1	51.33%	52.0006 %	32.1031 %	51.6811 %	51.7081 %
1	51.33%	52.0006 %	32.1031 %	51.6811 %	51.7081 %
10	51.33%	52.0006 %	32.1031 %	51.6811 %	51.7081 %
100	51.33%	52.0006 %	32.1031 %	51.6811 %	51.7081 %
1000	51.33%	52.0006 %	32.1031 %	51.6811 %	51.7081 %
10000	51.33%	52.0006 %	32.1031 %	51.6811 %	51.7081 %
100000	51.33%	52.0006 %	32.1031 %	51.6811 %	51.7081 %
1000000	50.25%	51.3333 %	25.2717 %	50.9314 %	50.944 %
10000000	50.48%	51.3333 %	25.2717 %	50.9314 %	50.944 %
100000000	50.48%	51.3333 %	25.2717 %	50.9314 %	50.944 %
1000000000	50.50%	51.3333 %	25.2717 %	50.9314 %	50.944 %

Table 4: Relative Absolute Prediction Error (memory leaks)

7.5.3 Mean Absolute Error

The Mean Absolute Error is the average of the differences between predicted and real remaining time to failure. Specifically, it is calculated as:

$$\frac{1}{n}\sum_{j=1}^{n}|P_j - T_j| \quad (8)$$

It is expressed in seconds, and the corresponding values are reported in Table 5.

Lambda Value	LASSO as a predictor	Linear Regression	M5P	SVM	SVM2
0.1	216.1371344	218.4609633	151.4749032	217.2030431	217.1994416
1	216.1371344	218.4609633	151.4749032	217.2030431	217.1994416
10	216.1371344	218.4609633	151.4749032	217.2030431	217.1994416
100	216.1371344	218.4609633	151.4749032	217.2030431	217.1994416
1000	216.1371344	218.4609633	151.4749032	217.2030431	217.1994416
10000	216.1371343	218.4609633	151.4749032	217.2030431	217.1994416
100000	216.1371328	218.4609633	151.4749032	217.2030431	217.1994416
1000000	211.5652009	215.6813791	122.3348313	213.9885213	213.9984200
10000000	212.5240639	215.6813611	122.2207137	213.9952387	213.9901230
100000000	212.5200248	215.6813611	122.2207137	213.9952387	213.9901230
1000000000	212.6286891	215.6813611	122.2207137	213.9952387	213.9901230

Table 5: Mean Absolute Error (memory leaks)

7.5.4 Soft-Mean Absolute Error

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

Tolerance threshold: 10%

In this case, if $|P_j - T_j| < 0.1\,T_j$ then $|P_j - T_j|$ is assumed to be equal 0. In Table 6, values are given in seconds.

Lambda Value	LASSO as a predictor	Linear Regression	M5P	SVM	SVM2
0.1	205.2793103	210.4197570	138.5883843	209.7832957	209.7901635
1	205.2793103	210.4197570	138.5883843	209.7832957	209.7901635
10	205.2793103	210.4197570	138.5883843	209.7832957	209.7901635
100	205.2793103	210.4197570	138.5883843	209.7832957	209.7901635
1000	205.2793103	210.4197570	138.5883843	209.7832957	209.7901635
10000	205.2793101	210.4197570	138.5883843	209.7832957	209.7901635
100000	205.2793079	210.4197570	138.5883843	209.7832957	209.7901635

1000000	199.8597987	207.6079917	106.0350537	206.2501818	206.2755247
10000000	200.9452037	207.6079738	105.9163482	206.2563593	206.2507162
100000000	200.9398692	207.6079738	105.9163482	206.2563593	206.2507162
1000000000	201.0012186	207.6079738	105.9163482	206.2563593	206.2507162

Table 6: Soft-Mean Absolute Error (10% tolerance, memory leaks)

Tolerance threshold: 5 minutes (300 seconds)

In this case, if $|P_j - T_j| < 300$ then $|P_j - T_j|$ is assumed to be equal 0. In the following table, values are given in seconds.

Lambda Value	LASSO as a predictor	Linear Regression	M5P	SVM	SVM2
0.1	134.0253637	132.6208366	66.86037669	135.0547981	134.8198715
1	134.0253637	132.6208366	66.86037669	135.0547981	134.8198715
10	134.0253637	132.6208366	66.86037669	135.0547981	134.8198715
100	134.0253637	132.6208366	66.86037669	135.0547981	134.8198715
1000	134.0253637	132.6208366	66.86037669	135.0547981	134.8198715
10000	134.0253637	132.6208366	66.86037669	135.0547981	134.8198715
100000	134.0253632	132.6208366	66.86037669	135.0547981	134.8198715
1000000	131.8484762	132.5387803	44.90684454	134.6272875	134.5863147
10000000	132.2742505	132.5387800	44.70903621	134.6537407	134.5866406
100000000	132.2738022	132.5387800	44.70903621	134.6537407	134.5866406
1000000000	131.7926044	132.5387800	44.70903621	134.6537407	134.5866406

Table 7: Soft-Mean Absolute Error (5 minutes tolerance, memory leaks)

7.5.5 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 8.

Lambda Value	Linear Regression	M5P	SVM	SVM 2
0.1	0.10	1.75	183.68	202.03
1	0.12	1.27	193.95	214.11
10	0.11	1.36	177.64	202.84
100	0.09	1.21	182.07	210.77
1000	0.12	1.17	179.25	198.59
10000	0.11	1.15	177.05	193.04
100000	0.10	1.17	200.49	238.97
1000000	0.09	1.24	211.34	250.61
10000000	0.12	1.14	182.31	237.16
100000000	0.10	1.13	201.02	213.90
1000000000	0.09	1.16	186.91	218.40

Table 8: WEKA Training Time (memory leaks)

7.5.6 Validation Time

The Validation Time represents the time needed to validate the accuracy of a model. It is expressed in seconds in Table 9.

Lambda Value	Linear Regression	M5P	SVM	SVM 2
0.1	0.09	0.07	0.07	0.07
1	0.09	0.07	0.07	0.07
10	0.09	0.07	0.07	0.07
100	0.08	0.07	0.07	0.07
1000	0.10	0.07	0.09	0.07
10000	0.09	0.07	0.08	0.07
100000	0.09	0.07	0.07	0.07
1000000	0.10	0.08	0.08	0.08
10000000	0.12	0.08	0.08	0.08
100000000	0.09	0.08	0.08	0.08
1000000000	0.10	0.08	0.08	0.08

Table 9: WEKA Validation Time (memory leaks)

7.5.7 Fitted Models

We report in the following Figures the fitted models for $\lambda = 10^9$ using all the WEKA training models plus Lasso. 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, while on the y-axis we have the *predicted* RTTC.

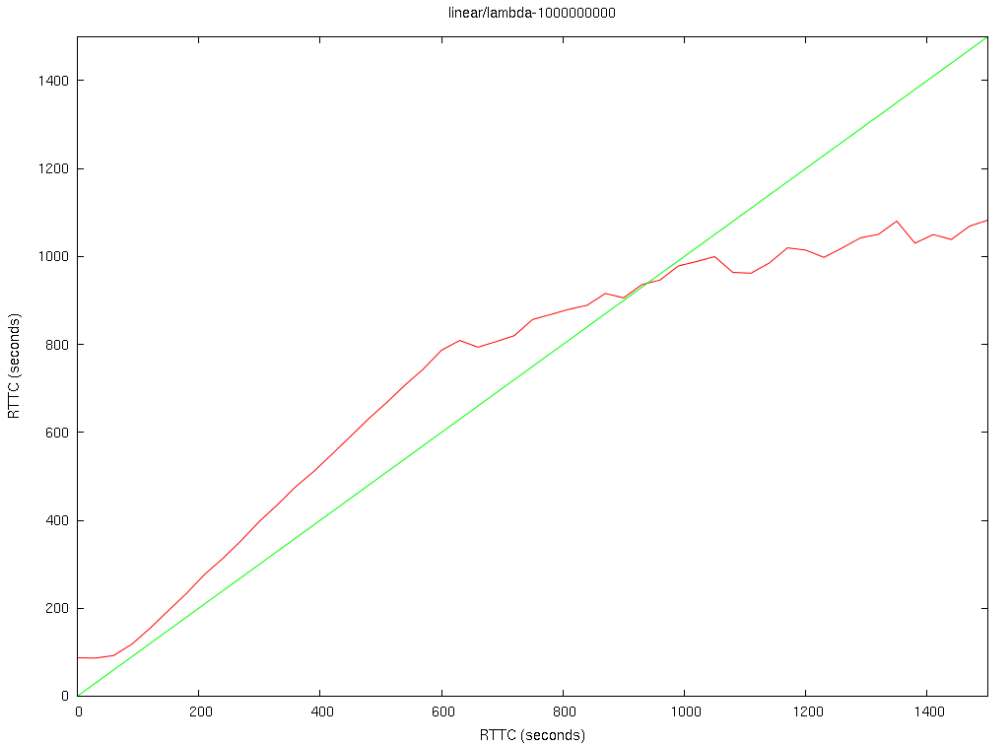

Figure 17: Prediction with Linear Regression

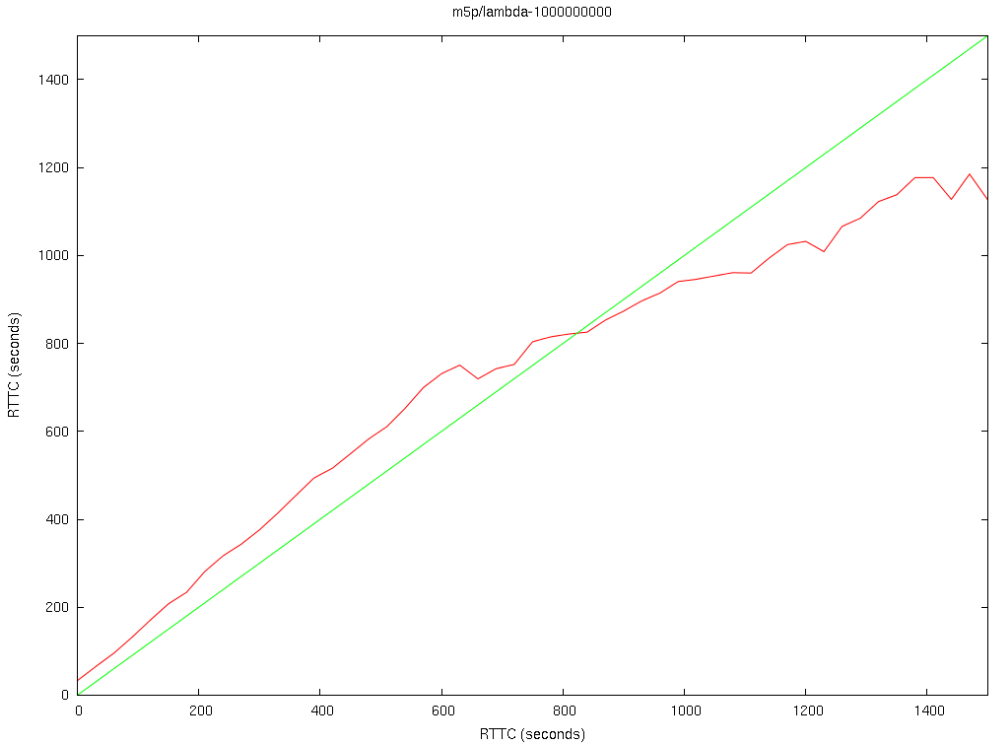

Figure 18: Prediction with MP5

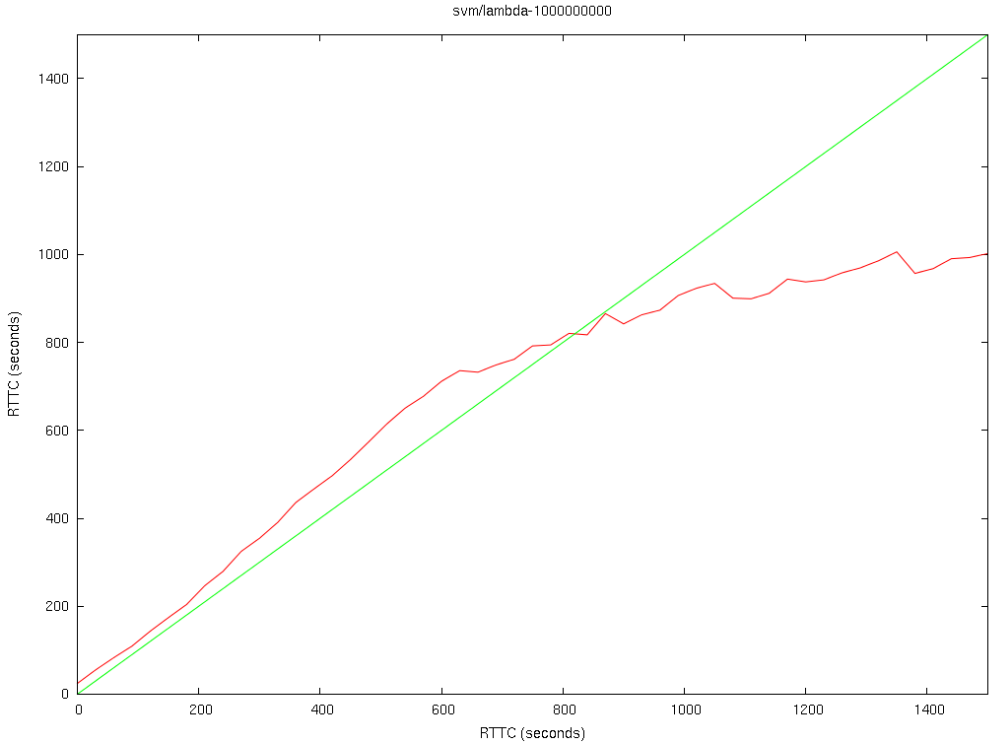

Figure 19: Prediction with SVM

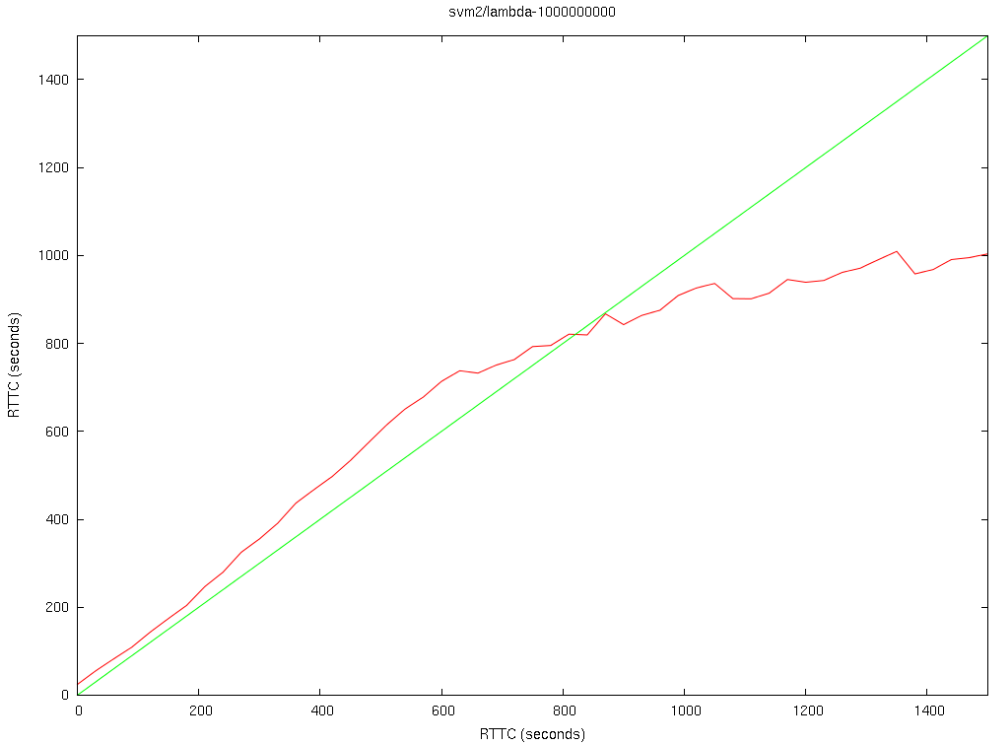

Figure 20: Prediction with SVM2

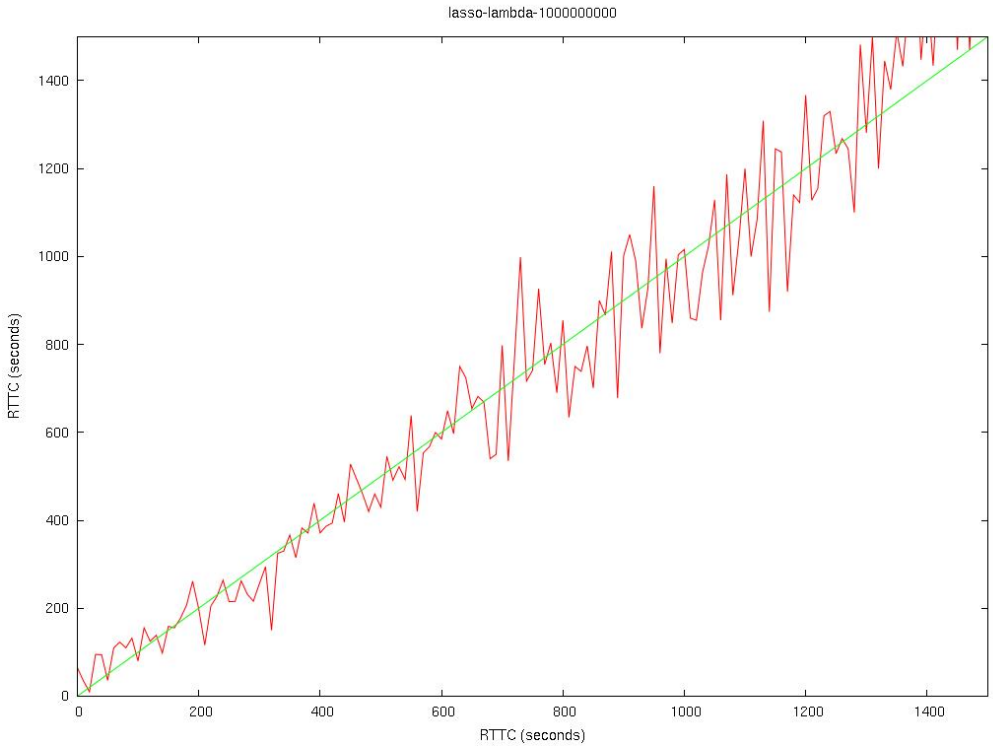

Figure 21: Prediction with Lasso as a Predictor

The selection of the ML method can be done based on the prediction errors, training time, validation time and predicting the RTTC in run time.

8 MACHINE LEARNING FRAMEWORK: PREDICTION WITH MEMORY LEAKS, UNTERMINATED THREADS, AND 64 TPC-W CLIENTS

We have collected data related to the behaviour of the TPC-W server when injecting memory leaks 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, and we report in this section the outcome of our learning framework.

8.1 Behaviour of the System

Figure 22 shows the system parameters (before feeding them into Lasso) on a large scale machine (HP ProLiant server), with an increased amount of computing power. The dotted curves (cpu usage) are to be read against the right-hand side y axis, while the solid ones (memory usage) are to be read against the left-hand size y axis.

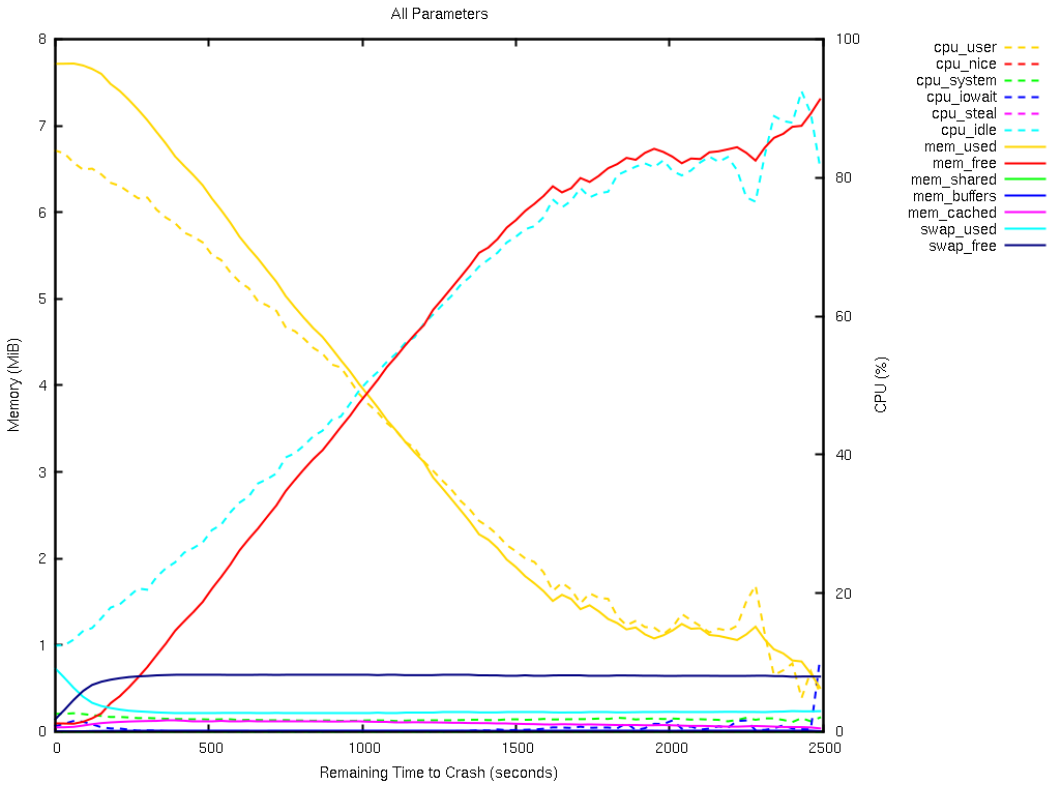

Figure 22: System parameters (memory leaks and unterminated threads)

It is clear by the results that the HP ProLiant server provides, due to the increased power, more stable results which are affected only by memory leaks. In this scenario, Lasso will select less parameters even with small values of lambda, and the weights chosen by Lasso will be smaller.

8.2 TPC-W Response Time

Figure 23 presents the Response Time of the TPC-W web application when injecting memory leaks and unterminated threads in the system. Specifically, this is an average of the response time as seen by all the 64 clients.

Initially, the response time is very reduced (never higher than 0.5 seconds, usually around 0.2 seconds). When the system starts to suffer from memory leakage and unterminated threads (around 1000 seconds) the response time increases, but stays nevertheless on a sustainable value (less than 1 second). When the system starts to swap data out of main memory (around 1200 seconds), the response time increases over an acceptable threshold, reaching a value of 2.5 seconds just before the system crash.

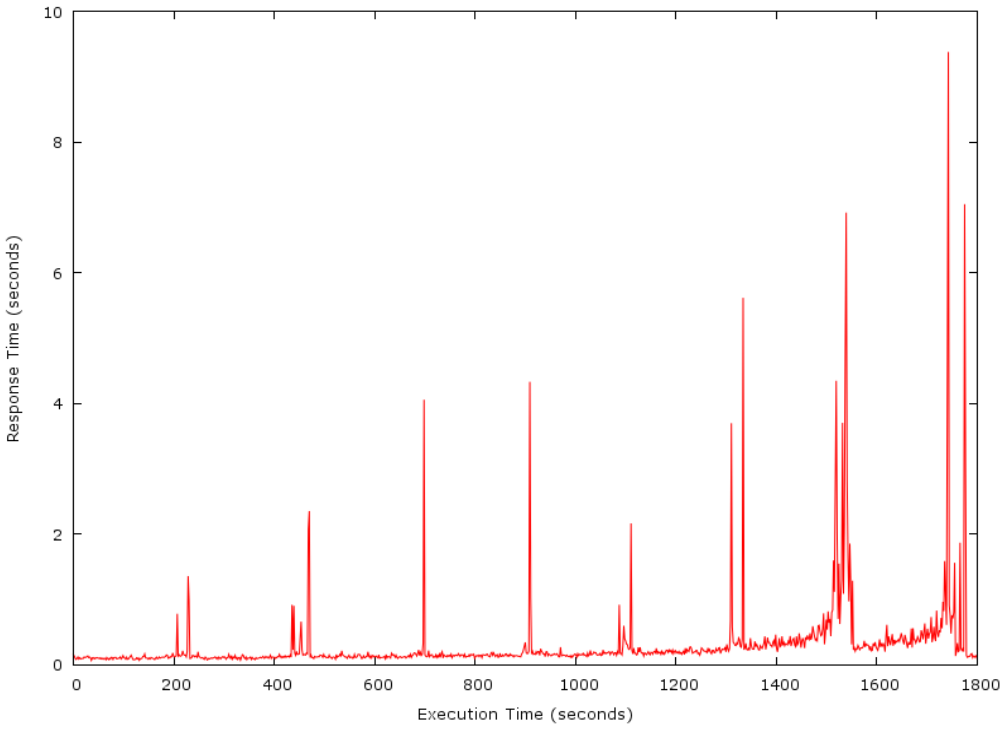

Figure 23: TPC-W (64 clients) Response time (memory leaks and unterminated threads)

8.3 Parameters Selected by Lasso

The amount of selected parameters according to the different values of λ is reported in Table 10.

λ	# Parameters		λ	# Parameters
0.1	2		100000	2
1	2		1000000	3
10	2		10000000	2

100	2		100000000	2
1000	2		1000000000	2
10000	2			

Table 10: Parameters Selected by Lasso (memory leaks and unterminated threads)

In Table 11, we show the weights of parameters selected by Lasso for values of λ providing minimum prediction error (i.e. for λ between 10^6 and 10^9).

$\lambda = 1000000$	mem_used	0.000019235560086
	mem_free	0.000236946638676
	swap_free	0.000263386541515
$\lambda = 10000000$	mem_used	0.000019235737077
	mem_free	0.000236946401283
$\lambda = 100000000$	mem_used	0.000019235737077
	mem_free	0.000236946401283
$\lambda = 1000000000$	mem_used	0.000000265373490
	mem_free	0.000212720823984

Table 11: Weights assigned by Lasso, when injecting memory leaks and unterminated threads

8.4 Trend of Parameters selected by Lasso

The following chart show variations of input features averaged for all executed runs, when selecting $\lambda = 10^6$.

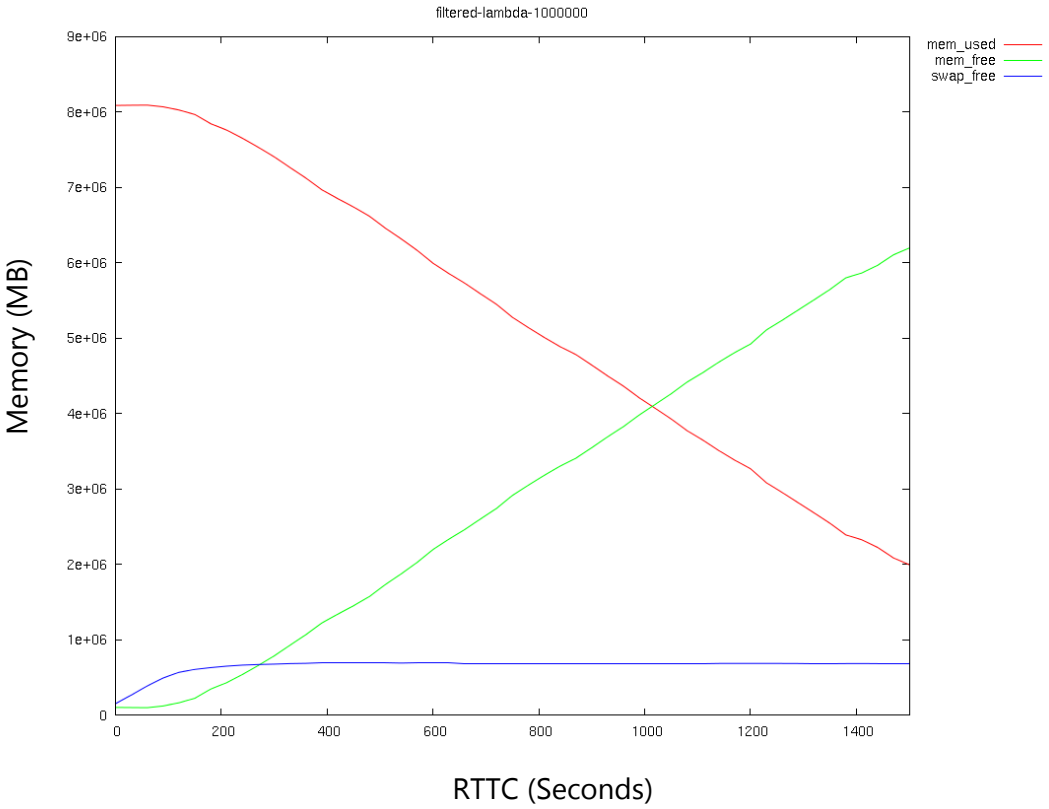

Figure 24: Parameters Selected by Lasso (memory leaks and unterminated threads, $\lambda = 10^6$)

8.5 Machine Learning Model Building

Once calculated values of the vector β for each value of λ, WEKA is used for building machine learning models for predicting the Remaining Time To Failure (RTTF) of the system. We used three learning methods, i.e. Linear Regression, M5P (decision tree with the possibility of linear regression functions at the nodes) and Supported Vector Machine (SVM). Before building learning models, the script `applyBeta` is executed for producing input files to WEKA. The script filters out all features for which the calculated weight of the vector β is equal to zero. Thus, the remaining features are used for training the machine learning models through the three above-mentioned learning methods. In the following, we show experimental data related to different learning algorithms.

8.5.1 Maximum Absolute Prediction Error

This error represents the highest error (worst case) encountered during the prediction. It is expressed in seconds, and the corresponding values are reported in Table 3.

Lambda Value	LASSO as a predictor	Linear Regression	M5P	SVM	SVM2
0.1	662.2829868	133.3580	131.4476	131.5915	131.5920
1	662.2829868	133.3580	131.4476	131.5915	131.5920

10	662.2829868	133.3580	131.4476	131.5915	131.5920
100	662.2829868	133.3580	131.4476	131.5915	131.5920
1000	678.4059344	131.4770	129.1498	130.4038	130.4033
10000	662.2829869	133.3580	131.4476	131.5915	131.5920
100000	662.2829880	133.3580	131.4476	131.5915	131.5920
1000000	664.4462479	130.4167	116.4080	128.3623	128.3636
10000000	662.2831096	133.3580	131.4476	131.5915	131.5920
100000000	662.2842154	133.3580	131.4476	131.5915	131.5920
1000000000	662.2952727	133.3580	131.4476	131.5915	131.5920

Table 12: Maximum Absolute Prediction Error (memory leaks and unterminated threads)

8.5.2 Relative Absolute Prediction Error

The relative absolute E_i 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.

$$E_i = \frac{\sum_{j=1}^{n}|P_j - T_j|}{\sum_{j=1}^{n}|T_j - \overline{T}|} \quad (9)$$

Where P_j is the value predicted for sample j (out of n samples), $T_j$is the target (i.e., ground truth) value for sample j, and:

$$\overline{T} = \frac{1}{n}\sum_{j=1}^{n} T_j \quad (10)$$

It is expressed in percentage, and the corresponding values are reported in Table 13.

Lambda Value	LASSO as a predictor	Linear Regression	M5P	SVM	SVM2
0.1	30,43%	28.6613 %	28.2508 %	28.2828 %	28.2818 %
1	30,43%	28.6613 %	28.2508 %	28.2828 %	28.2818 %
10	30,43%	28.6613 %	28.2508 %	28.2828 %	28.2818 %
100	30,43%	28.6613 %	28.2508 %	28.2817 %	28.2818 %
1000	30,33%	28.5928 %	28.0867 %	28.3594 %	28.3593 %
10000	30,43%	28.6613 %	28.2508 %	28.2817 %	28.2818 %
100000	30,43%	28.6613 %	28.2508 %	28.2817 %	28.2818 %

1000000	29,71%	28.0292 %	25.0184 %	27.5877 %	27.5879 %
10000000	30,43%	28.6613 %	28.2508 %	28.2817 %	28.2818 %
100000000	30,43%	28.6613 %	28.2508 %	28.2817 %	28.2818 %
1000000000	30,43%	28.6613 %	28.2508 %	28.2817 %	28.2818 %

Table 13: Relative Absolute Prediction Error (memory leaks and unterminated threads)

8.5.3 Mean Absolute Error

The Mean Absolute Error is the average of the differences between predicted and real remaining time to failure. Specifically, it is calculated as:

$$\frac{1}{n}\sum_{j=1}^{n}\left|P_j - T_j\right| \quad (11)$$

It is expressed in seconds, and the corresponding values are reported in Table 14.

Lambda Value	LASSO as a predictor	Linear Regression	M5P	SVM	SVM2
0.1	143.6199729	133.3675869	131.5062473	131.6043759	131.6055948
1	143.6199729	133.3675869	131.5062473	131.6043759	131.6055948
10	143.6199729	133.3675869	131.5062473	131.6043759	131.6055948
100	143.6199729	133.3675869	131.5062473	131.6043759	131.6055948
1000	143.1646206	131.5067840	129.2244036	130.4455055	130.4376640
10000	143.6199729	133.3675869	131.5062473	131.6043759	131.6055948
100000	143.6199728	133.3675869	131.5062473	131.6043759	131.6055948
1000000	140.2498030	130.4311705	121.5834724	128.3749487	128.3819701
10000000	143.6199693	133.3675869	131.5062473	131.6043759	131.6055948
100000000	143.6199373	133.3675869	131.5062473	131.6043759	131.6055948
1000000000	143.6196173	133.3675869	131.5062473	131.6043759	131.6055948

Table 14: Mean Absolute Error (memory leaks and unterminated threads)

8.5.4 Soft-Mean Absolute Error

The Soft-Mean Absolute Error is calculated as the Mean Absolute Error except that when the value $\left|P_j - T_j\right|$ is below a given threshold it is assumed to be equal to 0.

Tolerance threshold: 10%

In this case, if $|P_j - T_j| < 0.1\,T_j$ then $|P_j - T_j|$ is assumed to be equal 0. In Table 15, values are given in seconds.

Lambda Value	LASSO as a predictor	Linear Regression	M5P	SVM	SVM2
0.1	123.6696959	111.9533306	110.0307302	110.2644298	110.2158174
1	123.6696959	111.9533306	110.0307302	110.2644298	110.2158174
10	123.6696959	111.9533306	110.0307302	110.2644298	110.2158174
100	123.6696959	111.9533306	110.0307302	110.2644298	110.2158174
1000	122.7723825	109.7365837	107.7403929	109.4491650	109.4333442
10000	123.6696959	111.9533306	110.0307302	110.2644298	110.2158174
100000	123.6696959	111.9533306	110.0307302	110.2644298	110.2158174
1000000	120.2688198	108.9415605	99.8727128	107.1623191	107.1376641
10000000	123.6696950	111.9533306	110.0307302	110.2644298	110.2158174
100000000	123.6696867	111.9533306	110.0307302	110.2644298	110.2158174
1000000000	123.6696042	111.9533306	110.0307302	110.2644298	110.2158174

Table 15: Soft-Mean Absolute Error (10% tolerance, memory leaks and unterminated threads)

Tolerance threshold: 5 minutes (300 seconds)

In this case, if $|P_j - T_j| < 300$ then $|P_j - T_j|$ is assumed to be equal 0. In the following table, values are given in seconds.

Lambda Value	LASSO as a predictor	Linear Regression	M5P	SVM	SVM2
0.1	44.56055406	37.38165886	36.46860749	40.59226086	40.62698578
1	44.56055406	37.38165886	36.46860749	40.59226086	40.62698578
10	44.56055406	37.38165886	36.46860749	40.59226086	40.62698578
100	44.56055406	37.38165886	36.46860749	40.59226086	40.62698578
1000	45.56608711	32.58955594	31.47044948	34.61567365	34.78775631
10000	44.56055406	37.38165886	36.46860749	40.59226086	40.62698578
100000	44.56055408	37.38165886	36.46860749	40.59226086	40.62698578
1000000	44.03004684	36.49058831	29.56826065	39.29803625	39.38115168
10000000	44.56055607	37.38165886	36.46860749	40.59226086	40.62698578
100000000	44.56057418	37.38165886	36.46860749	40.59226086	40.62698578
1000000000	44.56075525	37.38165886	36.46860749	40.59226086	40.62698578

Table 16: Soft-Mean Absolute Error (5 minutes tolerance, memory leaks and unterminated threads)

8.5.5 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 17.

Lambda Value	Linear Regression	M5P	SVM	SVM 2
0.1	0.16	2.42	499.26	567.88
1	0.20	2.96	470.44	544.71
10	0.16	3.18	490.52	542.57
100	0.17	2.82	464.01	518.46
1000	0.18	2.05	470.67	550.43
10000	0.17	2.71	457.84	516.10
100000	0.17	2.71	464.86	560.96
1000000	0.17	2.58	515.35	487.63
10000000	0.17	2.67	462.85	521.80
100000000	0.17	2.66	457.03	536.37
1000000000	0.16	2.65	471.56	528.22

Table 17: WEKA Training Time (memory leaks and unterminated threads)

8.5.6 Validation Time

The Validation Time represents the time needed to validate the accuracy of a model. It is expressed in seconds in Table 18.

Lambda Value	Linear Regression	M5P	SVM	SVM 2
0.1	0.12	0.12	0.13	0.12
1	0.20	0.11	0.10	0.11
10	0.12	0.17	0.10	0.11
100	0.13	0.17	0.10	0.10
1000	0.16	0.15	0.15	0.15
10000	0.13	0.13	0.10	0.10
100000	0.13	0.13	0.10	0.11
1000000	0.14	0.15	0.15	0.12
10000000	0.13	0.12	0.10	0.10
100000000	0.13	0.13	0.10	0.10
1000000000	0.13	0.13	0.10	0.10

Table 18: WEKA Validation Time (memory leaks and unterminated threads)

8.5.7 Fitted Models

We report in the following Figures the fitted models for $\lambda = 10^6$ using all the WEKA training models plus Lasso. 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, while on the y-axis we have the *predicted* RTTC.

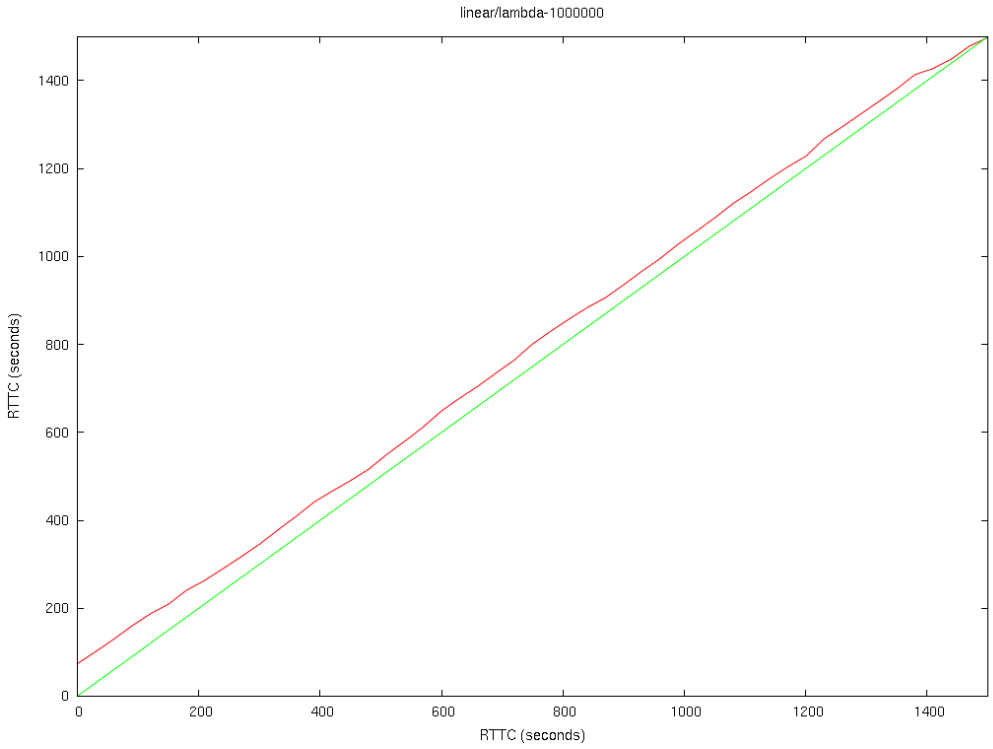

Figure 25: Prediction with Linear Regression

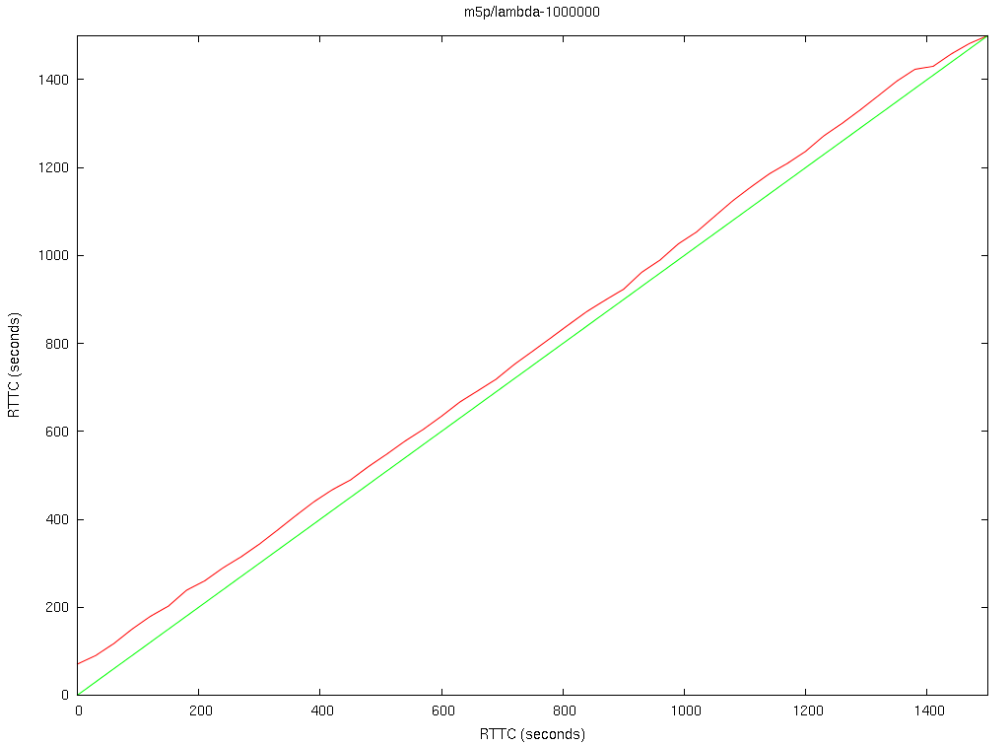

Figure 26: Prediction with M5P

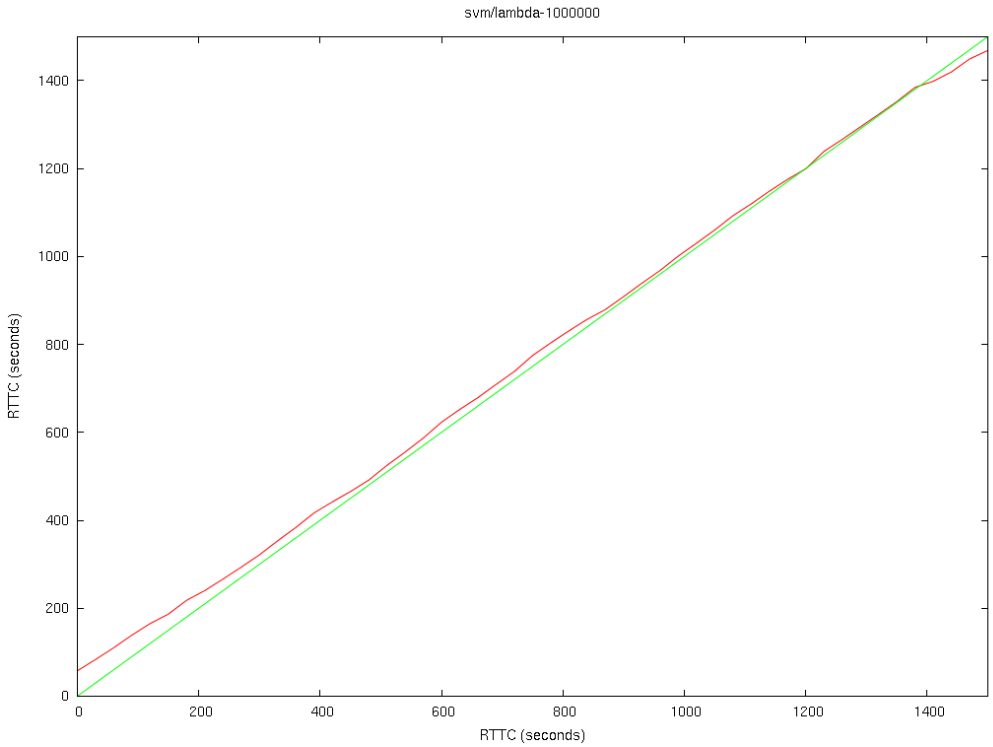

Figure 27: Prediction with SVM

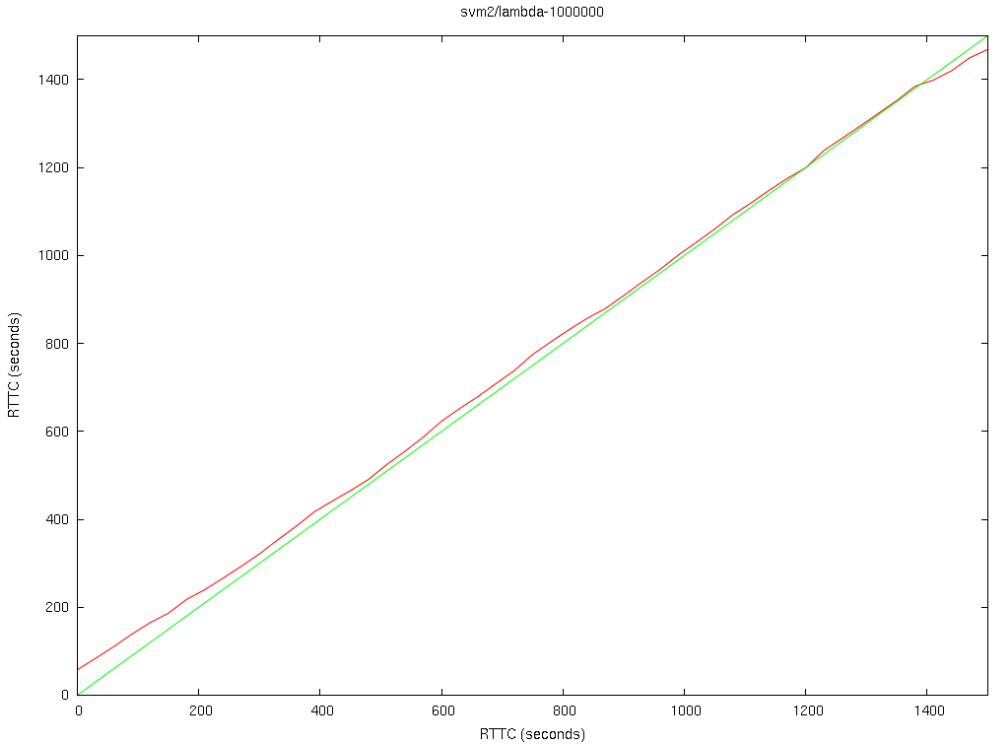

Figure 28: Prediction with SVM2

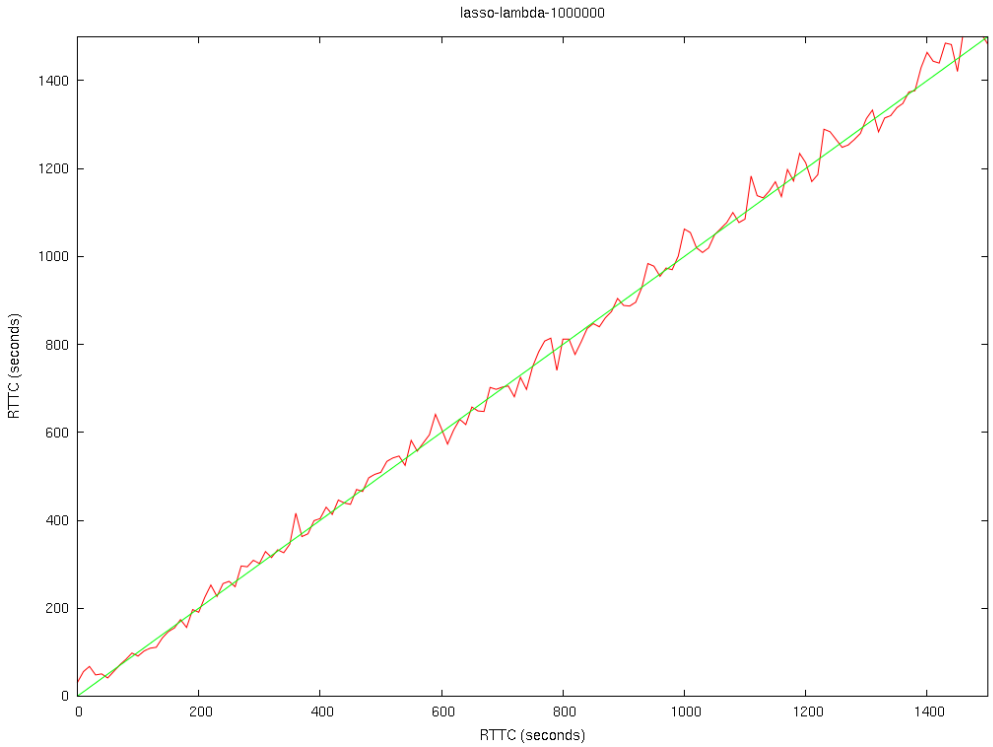

Figure 29: Prediction with LASSO

9 DISCUSSION ON THE LEARNED MODELS

The results in Sections 7 and 8 for both scenarios with 64 TPC-W clients (memory leaks and memory leaks jointly with unterminated threads) highlights prediction models, which have the smallest training time and the best accuracy for the given input data. The prediction models obtained by M5P algorithm are preferable, since they provide smaller prediction errors and acceptable training time.

Generally speaking, higher values of λ provide smaller prediction errors. In the case of memory leaks only, this is true for $\lambda = 10^9$, while for memory leaks and unterminated threads the smallest prediction error is given by $\lambda = 10^6$. This is compatible with the linear regression performed by Lasso, if we take into account that the memory leaks and unterminated threads injection rate used for experimenting with the ML Framework produce anomalies trends which can be easily used for predicting the RTTC, which is a linear function. In fact, the average over a very high number of executions (as we have done for collecting experimental data) provide trends which can be easily converted into a linear rule (this can be clearly seen in Figure 14 and Figure 22). The last part of experimentation of this framework, as we will discuss in Section 10, deals with online prediction of the RTTC to enforce prompt proactive rejuvenation of about-to-fail VMs. In the first part of the experimentation we will rely on Lasso as a predictor to perform the prediction.

Lasso as a predictor, as shown by the results in Sections 7 and 8, does not provide a prediction error as low as MP5. However, we use it due to its simplicity and quick responsiveness in the prediction, which is an important aspect to support a prompt rejuvenation of the system.

In the next part of the experimentation, we will use prediction models by the other learning algorithms to assess their performance impact on the prediction. In this way, the selection of the best-suited prediction model will be driven by both accuracy and online performance, which is essential for the final goal of our project.

10 DYNAMIC RECONFIGURATION BASED ON ML PREDICTION

The models generated during the phases, described in Sections (3, 7 and 8), are used to predict the RTTC. In this part of the experimentation, we used 3 Virtual Machines installed on VMware Workstation; VM1 (controller) will be in charge of monitoring the set of parameters (CPU, Memory, Swap, ...) collected on VM2 (client) and VM3 (client). The Proactive Control Module in the controller evaluates one of the prediction models obtained by the ML Framework.

TPC-W Users (Emulated Browser, as described in Section 6) is installed on an additional virtual machine, which is connected to the virtualized set of VMs relying on a packet forwarder (a simplified version of a load balancer) installed on VM1. As well, the Proactive Control Module allows transparently to reach the currently active VM

10.1 Dynamic Reconfiguration Flow Diagram

Figure 30 shows the behaviour of the system when running in a Soft Rejuvenation mode. 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).

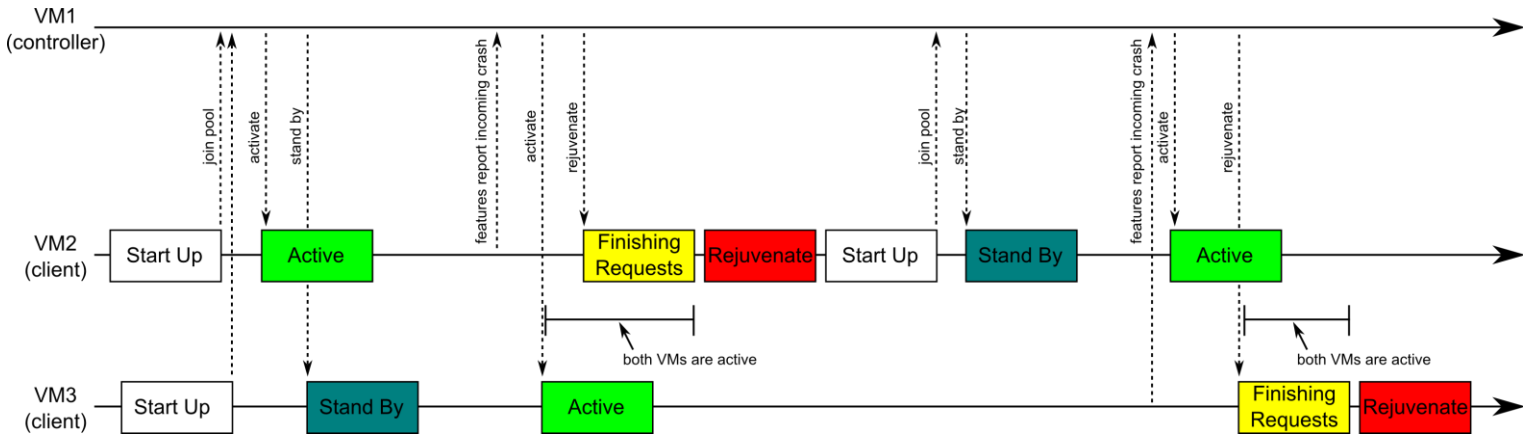

Figure 30: Dynamic Reconfiguration Flow Diagram (Soft Rejuvenation)

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, when TPC-W users start sending

requests for processing web interactions, which forwarded to the current active VM.

FMCs on both VM1 and VM2 then start sending performance measurements to the PCMC. 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.

10.2 Experimental Results: Response Time (as seen by TPC-W Clients) variation

The Average System Response Time and the System Features, in the presence memory leaks and unterminated threads (without rejuvenation), are presented in Figure 31. The response time increases exponentially in the period of time, when system is close to fail.

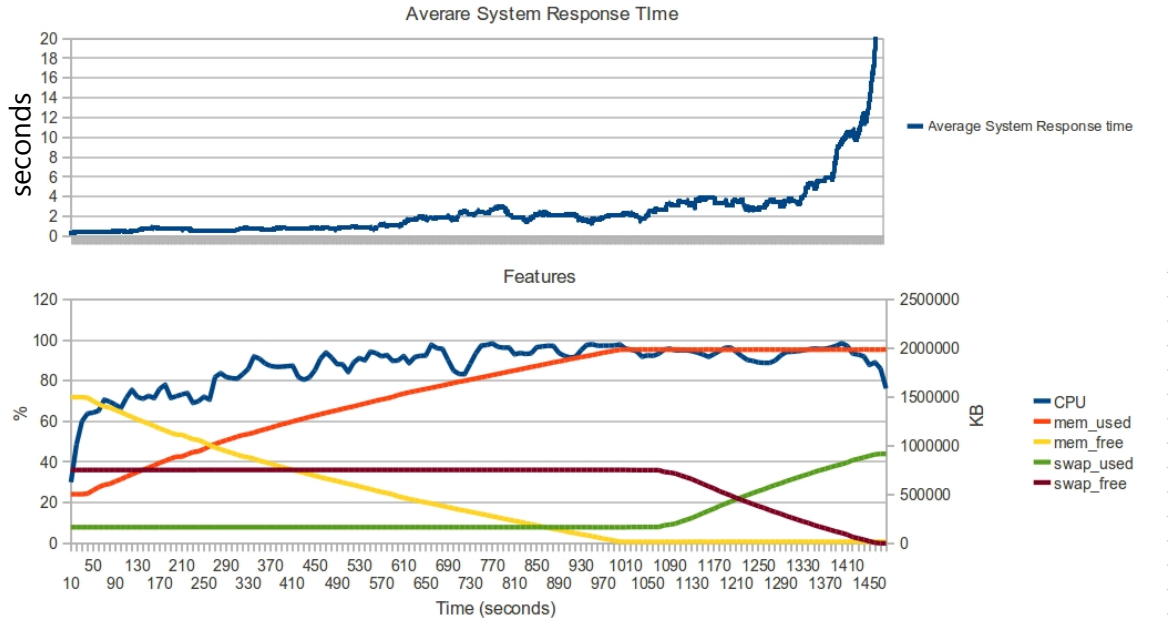

Figure 31: Average System Response Time and System Features without rejuvenation

The average system response times with and without rejuvenation are shown in Figure 32. We conducted experiments with 64 clients and in the presence of memory leaks and unterminated threads. Results show that through ML Framework important properties can be achieved, such as Seamless Execution, Self-Rejuvenation, Self-Reconfiguration and higher

[1] This prediction is done by using the prediction model generated by ML Framework. The term *about* should be read as *the RTTC 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.

Availability. As well, the virtualization and ML Framework guarantees the response time to be below a threshold, equal to 3 seconds.

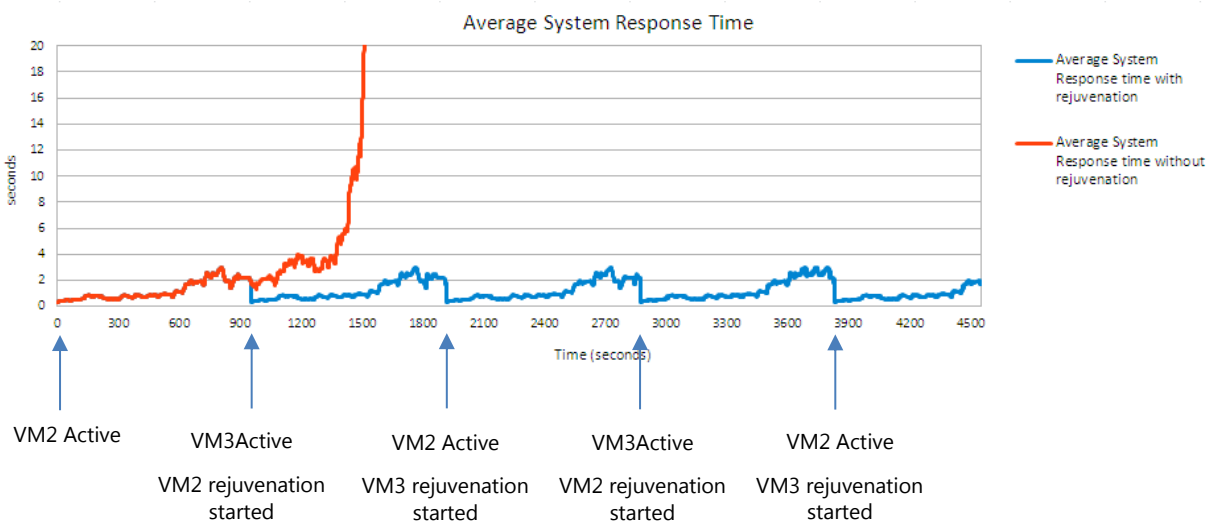

Figure 32: Average System Response Time Comparison with and without Rejuvenation

11 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 reliable 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 during the run time.

The proposed Virtualization Framework can be used for forming Private Clouds and controlled by the Intra-Autonomic Cloud Managers (Intra-ACM). They are controlled by the Inter-Autonomic Cloud Manager (Inter-ACM). In this way, Inter-ACM can monitor and govern the behaviour of the Federated Clouds and assure their autonomic properties.

12 APPENDIX: DATA ANALYTICS AS A SERVICE USE CASE

This appendix presents preliminary results, related to the Use Case (UC) 2, that can be utilized for Validation of the developed Machine Learning framework in the previous sections of this document.

Specifically, this appendix shows the progress made during this first year of the project to start preparing the DAaaS UC 2 to be used by the ML Framework in PANACEA. Mainly, the work focuses on studying the deployment requirements of the DAaaS UC and on analyzing the different parameters that need to be monitored to successfully train the Machine Learning Framework.

12.1 Deployment of DAaaS UC

The objective is to identify the different components of the DAaaS software solution that will need to be monitored. Figure 33 represents the block diagram of the central DAaaS services.

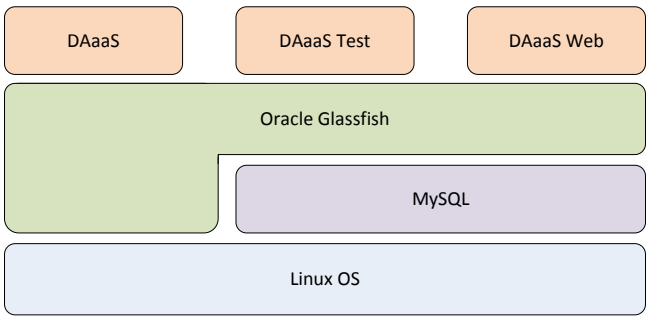

Figure 33: DAaaS block diagram

Apart from monitoring the Operating System, the DAaaS software solution relies on the MySQL database and on Oracle Glashfish application container. Any failure in both systems will bring the application down. DAaaS application itself is composed of the following components:

- DAaaS – The component that contains all the internal logic.
- DAaaS Test – A collection of tests to verify the installation of the application.
- DAaaS Web – Web interface to the user of the DAaaS application.

An Apache Hadoop [17] installation can be divided into two building groups: the software that controls all the map-reduce algorithm, and the software that performs the different tasks and stores the data. We can classify those components into a Master installation that will contain the HMaster (in case of using HBase storage), the JobTracker that assigns different tasks to the TaskTracker all around the Hadoop cluster and the NameNode that control the distribution of data all around the Hadoop Cluster (in case that we are using HDFS as storage solution).

Then we have a Slave node in the Hadoop cluster that it is going to perform the different map-reduce operations that it could be composed of a RegionServer (again, in case of using

HBase as storage solution), a TaskTracker and a DataNode (in case of using HDFS as storage solution). This is depicted in Figure 34.

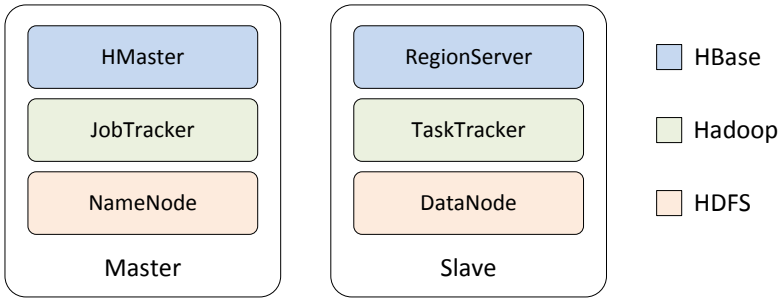

Figure 34: Apache Hadoop Block Diagram

Figure 35 represents a possible deployment example:

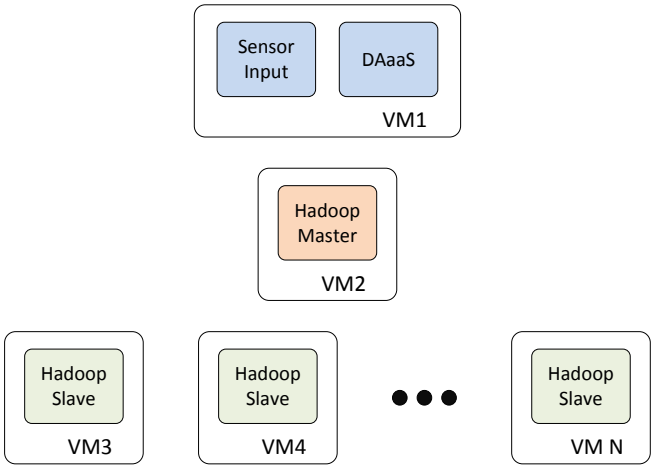

Figure 35: DAaaS deployment

In this case we will have three types of VMs:

- VM1 – In this VM the central DAaaS services will be installed, together with the "sensor input" module that will simulate measurements coming from a factory (see deliverable D4.1 [16] for more information).
- VM2 – That will contain the Hadoop Master (includes a NameNode and a JobTracker).
- VM3 to VM N – That will contain the necessary Hadoop Slaves: we can deploy as many as we want.

We need to get enough monitoring data from all those components to be able to train the ML Framework, although in the beginning we will focus on validating the solution for Hadoop.

12.1 Monitoring Metrics for DAaaS

In the deliverable D2.1 [18] the CollectD [19] monitoring solution has been selected for the PANACEA project. The following discussion focuses on how to use CollectD to monitor

DAaaS UC and get the necessary input data for the Machine Learning framework.

The different monitoring metrics will be presented divided into categories following the block diagrams presented in the previous section.

12.1.1 Monitoring the Operating System

All the VMs for the DAaaS UC will be using a Linux-based Operating System.

Metric	Description
Memory buffers	Size of memory buffers in MB
Cached memory	Size of cache memory in MB
CPU System Time	Amount of CPU time used by the system
CPU nice time	Amount of CPU time used by the users
CPU idle time	Amount of CPU time where the CPU is unused
CPU iowait time	Amount of CPU time the CPU has to wait for an input/output operation
Free disk space on /	Amount of free disk space in the root partition in MB
Free memory	Amount of free RAM memory in MB
Free swap	Amount of free SWAP memory in MB
Host boot time	Time for the host to boot in seconds
Host uptime	Time of the host since its last powerup/reboot in seconds
Incoming traffic lo	MB/secs incoming network traffic in lo network interface
Incoming traffic eth0	MB/secs incoming network traffic in eth0 network interface
# processes	Number of processes executed by the system at a given time
# running processes	Number of processes that are running in the system at a given time
# users connected	Number of users connected to the system at a given time
Outgoing traffic lo	MB/secs outgoing network traffic in lo network interface
Outgoing traffic eth0	MB/secs outgoing network traffic in eth0 network interface
Processor load	Percentage of usage of the CPU
Total disk space in /	Total disk space in root folder / in MB
Total memory	Total RAM memory in MB
Total swap space	Total SWAP memory size in MB
Used disk space /	Used disk space in MB
Used disk space %	Percentage of usage of the root disk
io	Number of input/output operations at kernel level

Table 19: Operating System Metrics

To collect those metrics we will make usage of CollectD plugins such as: CPU, Disk, Memory, etc.

12.1.2 Monitoring Glassfish

The DAaaS solution is a Java application that runs in the Oracle Glassfish container. The container needs to be carefully monitored. The following table presents the selected Glassfish parameters to be monitored:

Metric	Description
Threads used by the server	Number of threads used by the server
Memory used by the server	Size of the memory used by the Glassfish server in MB
Transactions	Number of transactions performed by the Glassfish server at a given time
Paranormal Activity – Queued Connections	Number of Queued Connections idle in the Glassfish server
Paranormal Activity – Errors	Number of errors detected by the Glassfish server
Paranormal Activity – Busy thread count	Number of threads that are being used for a long time
HTTP Sessions – active	Number of active http sessions opened to the server (it is related to the number of users at a given time)
HTTP Sessions – experied	Number of expired http sessions
JDBC Pools – Free	Number of free JDBC pools
JDBC Pools – Used Connections	Number of used connections to the different databases
JDBC Pools – Potential Connections	Number of possible JDBC leaks of not closed connections
JDBC Pools – Length of the wait queue	Number of the pending connections requests to a database

Table 20: Oracle Glassfish Metrics

Oracle Glassfish provides a REST interface to get the previously mentioned monitoring information. The plan is to use the CollectD Curl plugin to query that REST interface and be able to pull monitoring information from Classfish to CollectD.

12.1.3 Monitoring Apache Hadoop

Finally, we need to monitor all the Hadoop processes, the following table present a list of metrics to check the health of a Hadoop cluster:

Metric	Description
Total used space	In MB seeing from HDFS perspective
Total free space	In MB seeing from HDFS perspective
JobTracker – PREP	Number of tasks in preparation (total and %)

JobTracker – RUNNING	Number of tasks running (total and %)
JobTracker – COMPLETE	Number of tasks completed (total and %)
JobTracker – Killed	Number of tasks killed (total and %)
JobTracker – Failed	Number of tasks failed (total and %)
Number of MAP tasks	Number of map tasks at a given time
Number of REDUCE tasks	Number of REDUCE tasks at a given time
# total files in HDFS	Number of files in the Hadoop File System (HDFS)
Size of blocks in HDFS	Size of the blocks in the Hadoop File System (HDFS)
# Files created	Number of files created in the HDFS
# files listed	Number of files listed in the HDFS
# files renamed	Number of files renamed in the HDFS
# get block locations	Number of get block locations in HDFS
# blocks read	Number of read block operations in HDFS
# blocks written	Number of written blocks operations in HDFS
# blocks removed	Number of removed blocks operations in HDFS
# blocks replicated	Number of replicated blocks in HDFS
DataNode – # copy block ops	Number of copy block operations in an specific datanode
DataNode – Heartbeat	If the node responds to ping
DataNode - # read block ops	Number of read block operations in a datanode
DataNode - # replace block ops	Number of replace block operations in a datanode
DataNode - # write block ops	Number of write block operation in a datanode
DataNode - # checksum	Number of operations that verify the checksum of a block in a datanode
TaskTracker – Heartbeat	If the node responds to ping
TaskTracker – Operations	Number of operations in a specific tasktraker.

Table 21: Apache Hadoop Metrics

13 REFERENCES

[1] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" . In Proceedings of the USENIX Symposium on Internet Technologies and Systems, 2003.

[2] C. Cortes, and V. Vapnik, "Support-Vector Networks" . In Machine Learning, Kluwer Academic Publishers-Plenum Publishers,. DOI 10.1023/A:1022627411411, 1995.

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

[4] Ian H. Witten and Eibe Frank. "Data Mining: Practical Machine Learning Tools and Techniques" , Second Edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[5] Simon Perkins, Kevin Lacker, and James Theiler." *Grafting: fast, incremental feature selection by gradient descent in function space*" . J. Mach. Learn. Res. 3 (March 2003), 1333-1356, 2003.

[6] K. Vaidyanathan and K. S. Trivedi, "A comprehensive model for software rejuvenation," IEEE Trans. Dependable Secur. Comput., vol. 2, no. 2, pp. 124–137, 2005.

[7] G. A. Hoffmann, K. S. Trivedi, and M. Malek, "A best practice guide to resource forecasting for computing systems," IEEE Transactions on Reliability, vol. 56, no. 4, pp. 615–628, 2007.

[8] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," ACM Computing Surveys, 2010.

[9] L. Li, K. Vaidyanathan, and K. S. Trivedi, "An approach for estimation of software aging in a web server," in ISESE ' 02: Procs. of the Intl. Symp. on Empirical Software Engineering, 2002.

[10] 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 ACM conf. on Knowledge discovery and data mining, 2003.

[11] R. Vilalta, and A. Sivasubramaniam, "Critical event prediction for proactive management in large-scale computer clusters," in ACM conf. on Knowledge discovery and data mining, 2003.

[12] L. Cherkasova, K. M. Ozonat, N. Mi, J. Symons, and E. Smirni, "Anomaly? application change? or workload change? Towards automated detection of application performance anomaly and change." in IEEE/IFIP Intl. Conf. on Dependable Systems and Networks, 2008.

[13] I. Irrera, J. Dura andes, M. Vieira, and H. Madeira, "Towards identifying the best variables for failure prediction using injection of realistic software faults," in Dependable Computing (PRDC), IEEE Pacific Rim International Symposium on, 2010.

[14] Tutorial Proactive Fault Management for Availability Enhancement - Miroslaw Malek, Humboldt-Universitaet zu Berlin, Germany, Felix Salfner, Humboldt-Universitaet zu Berlin, Germany, IEEE DSN , 2009

[15] D. Simeonov and D. R. Avresky, "Proactive software rejuvenation based on machine learning techniques," in 1st Intl. Symp. on Cloud Computing. ICT, Munich, Germany, October 2009.

[16] Deliverable D4.1 of PANACEA project: Description of feasible use case. March 2014.

[17] Apache Hadoop http://hadoop.apache.org/, last visited September 30th, 2014.

[18] Deliverable D2.1 of PANACEA project: Principles of Pervasive Monitoring. March 2014.

[19] CollectD https://collectd.org/ , last visited September 30th, 2014.