SAPIENZA
UNIVERSITÀ DI ROMA

FACULTY OF ENGINEERING

Master's thesis in

MASTER OF SCIENCE IN ENGINEERING IN COMPUTER SCIENCE

Proactive Workload Management in Cloud Environments in the Presence of Software Aging

Candidate

Luca Forte

Thesis Advisor

Prof. Bruno Ciciani

Co-Advisors

Eng. Pierangelo Di Sanzo

Eng. Alessandro Pellegrini

Academic Year 2014/2015

Keep calm and...turn the power off then on!

Contents

Chapter 1

Introduction

The software aging is a well-known phenomenon that plagues many systems leading to degradation in performances and in higher response time[1]. In particular, it affects systems that have to run for a long period, due to the accumulation of the anomalies. These anomalies are often caused by errors in software development, as shown in [2]. They can be categorized into two classes of bugs: *Bohrbugs* and *Mandelbugs*[3]. The first ones are easily isolated and predictable while the latter ones are often unpredictable, and it is hard to reproduce them in a deterministic way, like the *aging-related bugs* as memory leaks, unterminated threads, unreleased locks and file fragmentation. The presence of these anomalies leads the system to fill all the free memory available for the processes, or to use more CPU than the needed, then leading the system either to an unusable condition or to work in a degradation mode. The main effect is a decrease in the system availability. Due to their nature, Mandelbugs are hard to seek and fix, so one of the better techniques to manage them is the *rejuvenation* [4]. It consists in leading the system in a clean-state, namely a state where the system is without bugs (or bugs occur with a low rate): a common rejuvenating procedure could be the reboot of the system. Rejuvenation is a management technique that can be executed either in a reactive manner or in a proactive manner, that is, in the first case when a fault is detected, then the rejuvenation procedure is triggered, while, on the other hand, it is possible to monitor some system features to predict when a system is leading towards a fault, acting in a user-chosen moment (e.g. when the workload is low/null). This work will rely on a Machine Learning(ML)-based framework for Proactive Client-server Application Management in the cloud, called PCAM[5], to perform a proactive software rejuvenation into a cloud region in a multi-cloud environment, where a distributed application is deployed. PCAM is able to predict the remaining time to the occurrence of some unexpected event on a virtual machine hosting a server instance, through optimized ML-based models product by a Framework for building Failure Prediction Models (F^2PM)[6] and continually measuring the system features. This work will present an innovative framework to perform a proactive workload management in multi-cloud environment exploiting the results in [6] and [5]. Respect to the two above mentioned works, this framework has to work with a distributed application on the cloud, where each cloud segment could be served by a different provider or a private infrastructure. Starting from previous results, this work will extend the previous intuitions to this more complicated scenario, showing how the system availability increases and the response time decreases when the framework is working on hybrid cloud systems too. To obtain the mentioned results, this work will exploit the *Mean Time To Failure* (MTTF) value, to decide which is the best cloud segment to serve an incoming request (i.e. the region with the lowest MTTF values) relying on the results provided by a leader in the system, and the *Remaining Time To Failure* (RTTF) to decide, into a cloud region, if a machine is approaching to a fault, then rejuvenate it to transparently switch the incoming traffic towards another active machine. Respect to previous works, this framework can be used with the most common applications, in which a user exploits the closest entry point to get a service by an application, but its request will be forwarded to a server in another geographical region. While previous works provide a method to predict the RTTF value for a virtual machine, an innovative balancing strategy will be used to forward the incoming traffic, using values computed by the leader (elected using the leader election algorithm presented in [7]) to define how many requests are supported by a particular region over the time. This framework was tested on a geographically distributed hybrid cloud, composed of VMs hosted on the Amazon EC2 service (in Ireland and in Frankfurt), and VMs hosted on a private server in Munich. The distributed application chosen to evaluate the framework is the TPC-W web benchmark[8].

The remainder of this work is structured as follows. In Chapter 2 it will be discussed related work. In Chapter 3 presents the problems related to the software aging. Chapter 4 discusses the ML-based frameworks used in this works. The design and the implementation choices of this framework are discussed Chapter 5. Chapter 6 shows the experimental evaluation and the Chapter 7 draws the conclusions.

Chapter 2

Related work

In the context of management of resources for cloud computing environments, several proposals have been presented in order to manage the accumulation of anomalies via proactive software rejuvenation [9]. Nevertheless, this work stands as an innovative solution, because it can manage any number of VMs, even geographically distributed, without any constraint on the topology of the distributed deploy of the application.

Specifically in the context of hybrid cloud environments, the work in [10] proposes a hybrid cloud computing model to make the best use of public cloud services along with privately-owned data centers. The paper presents as well a workload factoring service designed for proactive workload management. In [11], a resilient hierarchical distributed loop self-scheduling algorithm able to cope with VMs crashes is presented. Contrarily to these works, in this proposal it is enforced proactive rearrangement of the cloud organization, and it is further able to redirect incoming requests to different geographical areas, so as to reduce the impact of software rejuvenation of availability of the system.

A work similar in spirit to this proposal is the one in [12]. This paper proposes a capacity allocation algorithms which can coordinate multiple distributed resource controllers operating in geographically distributed cloud sites, coupled with a load redirection mechanism which distributes incoming requests among different sites. Nevertheless, the main focus of the proposal in [12] is to reduce the cost of allocated resources. In [13], custom interfaces for implementing policies and provisioning techniques for allocation of VMs under inter-networked Cloud computing scenarios are presented. Compared to these works, it is offered a transparent deploy of virtualized applications on a geographical scale, offering at the same time an increase in the availability of the system.

In [14], workload forecasting and optimal resource allocation is studied. This is done by illustrating a model-predictive algorithm for workload forecasting that is used for resource auto scaling. Similarly, the work in [15] presents a provisioning technique that automatically adapts to workload changes related to applications for facilitating the adaptive management of system and offering end-users guaranteed Quality of Services (QoS) in large, autonomous, and highly dynamic environments, using an analytic model. As well, in [16] Markovian Arrival Processes are used for the same purpose. Statistical models, for the same goal, are presented in [17]. Differently from these proposal, this work offered a complete framework which, being based on ML techniques, allows for the integration with any virtualized application. In particular, transparent deploy at a geographical scale, with self-tuning capabilities and proactive management of the workload are specific differences with these proposals.

In [18], the authors present a simulation framework for online capacity planning of cloud-based in-memory data stores. This work relies as well on ML methods, but only to determine network latency, while other aspects proper of the application are predicted using a simulative/analytic model. Contrarily, this work broaden the applicability of this proposal to any kind of application, not only in-memory data stores. Furthermore, this work relies only on ML techniques, which can capture hidden dynamics of the application.

Chapter 3

Software Aging

In computing systems, *Software Aging* is the name given to a phenomenon empirically observed in many software systems: as the runtime period of the system or process increases, its failure rate in a given time interval increases too. A failure may be an incorrect service (e.g. erroneous outcomes), no service (e.g. halt and/or crash of the system), or partial failure (e.g. increase in response time)[19]. These failures are caused by the presence of software faults. They can be categorized into two classes: *Bohrbugs* and *Mandelbugs*[3]. The first ones are easily isolated and manifest themselves consistently under well known conditions, typically it is simple to manage them. The latter ones show an apparent not-deterministic behavior, appearing chaotic, because of their fault activation. In most cases restarting a process it is possible that not recur the already seen faults. Because of their nature, generally they are difficult to manage.

A famous citation[1] help us to better understand of what we are talking about:

"Programs, like people, get old. We can't prevent aging, but we can understand its causes, take steps to limit its effects, temporarily reverse some

of the damage it has caused, and prepare for the day when the software is no longer viable.... (We must) lose our preoccupation with the first release and focus on the long term health of our products". [cit. D.L. Parnas]

In most cases, *Software Aging* affects systems that are running for a long period due to the accumulation of errors over the time, often related to software development, then to software faults. Natural consequence of it, is that after a certain time t, the system could be unusable for a user, in terms of response time and reliability, as shown in Figure 3.1, and if maintenance is needed, in term of system's availability too.

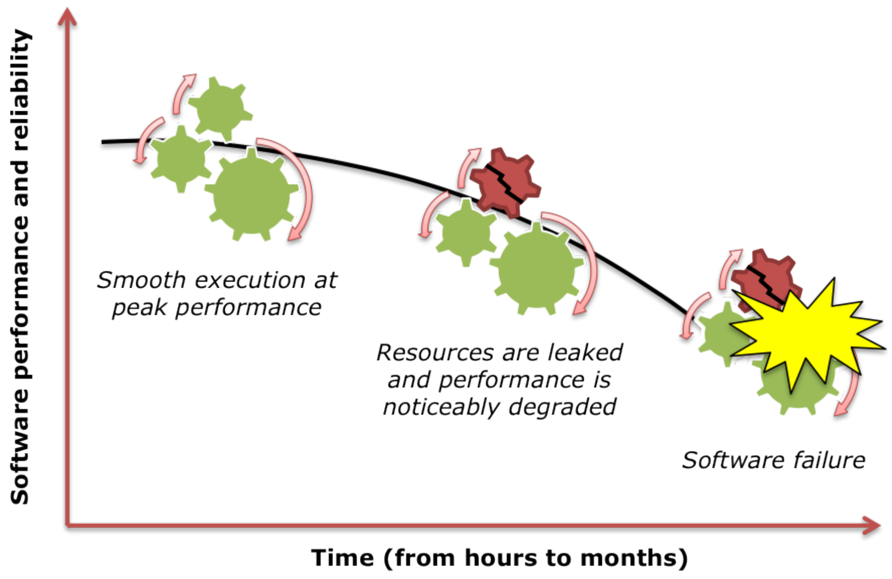

Software performance and reliability
Smooth execution at peak performance
Resources are leaked and performance is noticeably degraded
Software failure
Time (from hours to months)

Fig. 3.1: Degradation in performance over the time

Above mentioned maintenance could be the reboot of the system, it is just one of possible solutions to handle this problem, another way could be locate the guilty processes, that are causing performance degradation, and handle just them. Of course, a tradeoff exists, and often locate the processes and especially manage the code is not trivial even if it could avoid losing the current work session.

In the following section will be presented an overview on the main possible causes for the *Software Aging* problem. Again, some kinds of anomalies will be showed, focussing on their effects on the use of the operating system resources then on their effects on the system's availability.

3.1 Causes and effects

There are some kinds of faults that lead to degradation in performances of the system, they may be of different nature, but in this context, a particular kind of Mandelbugs are considered: the *aging-related bugs*. They are anomalies like *memory leaks*, *unreleased locks*, *unterminated threads*, *fragmentation*. Each of these anomaly produces, over the time, some effects to the system different from the effects produced by another one, but the occurrence of any of them, affects the system performances.

Just to provide a scenario of what an aging-related bugs could do, during the Gulf War, in February 1991, 25 US military were killed and 97 were injured when the Patriot missile-defense system in Dhahran, Saudi Arabia, failed to intercept an incoming Scud missile. It was due to a software fault in the system's weapons-control computer because of the increase in failure occurrence rate with the system runtime[20].

3.1.1 Anomalies

To better understand how these aberrations act, it will be provided a more detailed description of what they are.

- **Memory leak:** it incurs when a process incorrectly manages memory allocation. In particular, when memory that is no longer needed is not released (e.g. programming in C language, it could be happen missing a free call after a malloc call).

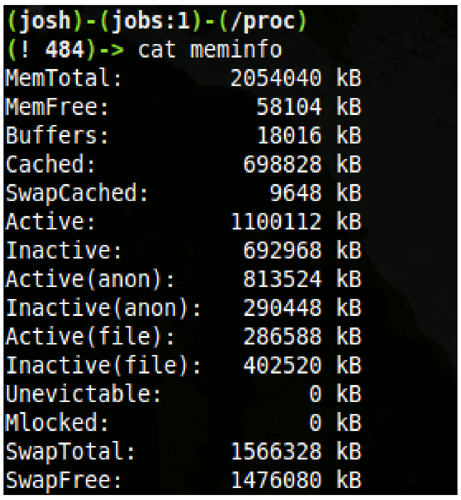
- **Unterminated thread:** it incurs when multi-thread programs need to start new threads never releasing them at the end of their life (e.g. programming in C language, it happens if a pthread exit call is forgotten at the end of the thread execution).

- **Unreleased lock:** it incurs when a process needs to work in a critical section. The natural way is to take the control of a shared memory area through lock primitives, then this lock could be released to allow the others to work, unfortunately it is not what always happens, causing this anomaly (e.g. programming in C language, it happens missing an unlock primitives after a lock call).

- **Fragmentation:** it is typical in system that are running for a long period. The objects tend to become spread out in the heap, forming a lot of small unused memory regions.

3.1.2 Effects

System's performances decrease when presented anomalies occur, as already seen in Figure 3.1. One of the affected components into the system is the memory. Due to the memory leaks, especially in machines that have to run for a long period, they cause the exhaustion of the free memory, forcing the system to resort in swapping operations. Tipically swap space is kept in secondary memory devices (e.g. disks) slower than the main memory, creating a bottleneck in memory access performing a lot of I/O operations. They lead the system to quickly reduce its performances when swap is needed, and the processes are not able to get free main memory to compute their routines. If the system is not able to reduce the memory used by current processes, the used swap memory will grow reducing the system's response time towards the total uselessness of it. Figure 3.2 shows when swap operations start due to the not available free memory.



```
(josh)-(jobs:1)-(/proc)
(! 484)-> cat meminfo
MemTotal:        2054040 kB
MemFree:           58104 kB
Buffers:           18016 kB
Cached:           698828 kB
SwapCached:         9648 kB
Active:          1100112 kB
Inactive:         692968 kB
Active(anon):     813524 kB
Inactive(anon):   290448 kB
Active(file):     286588 kB
Inactive(file):   402520 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:       1566328 kB
SwapFree:        1476080 kB
```

Fig. 3.2: Infos about OS's memory

Besides the memory problem just showed, also the CPU performances may be affected by the presented anomalies. It is proved that an average percentage of 40% of anomalies being due to errors in the software development[2]. One of the most frequent anomaly is the unterminated threads. It causes an increase in CPU use due to the schedule operation to perform threads. If a thread that had already finished its life is live yet, eventually it will be scheduled by the *Scheduler* even if it will have no operations to perform, causing wasting in CPU time for other threads that are waiting. This behavior causes an increase in the process response time. If there are a lot of hang threads, the response time could be exceed a threshold that bring the system to be unusable.

Even if the just presented anomalies are accused of being the main causes of software aging[3], unreleased locks and file fragmentation give their contribute to foster this problem. The first one causes an incorrect behavior in the process execution. If a shared memory area is locked, all the other processes that want to access that particular area, must wait until the lock is released. A bad manage of lock primitives could cause *deadlocks* in the worst case compromising the expected process execution. The latter anomaly is a well known problem in Operating Systems. In a nutshell files instead of to be stored continously in memory, are separated by little gaps, often too little to be fill with other files. Over the time, access became slower because of the files spreaded on the disk. Figure 3.3 reports an example of file fragmentation on disk.

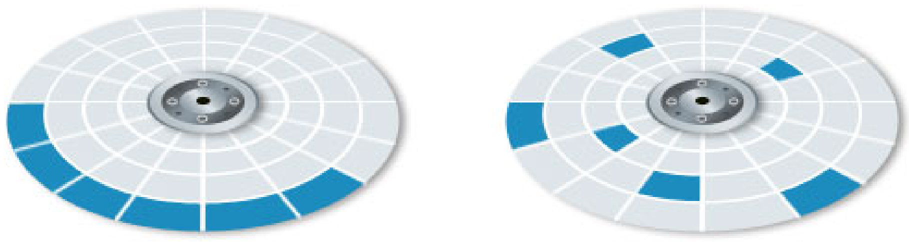


Fig. 3.3: Example of file fragmentation

3.1.3 Systems' availability

The discussed anomalies effects impact the system availability. First of all, it will provided a definition of what *Availability* is for a system, introducing *MTTF* and *MTTR* notions:

- **MTTF (Mean Time To Failure):** given a fixed time interval, it measures average time before a failure occurs.

- **MTTR (Mean Time To Repair):** given a fixed time interval, it measures average time to repair a faulty component.
- **Availability (A):** using the first two definitions, it represents the percentage value that the system is able to provide services given a fixed time interval, formally:

$$A = \frac{MTTF}{MTTF + MTTR} \tag{3.1}$$

The Figure 3.4 shows MTTF and MTTR over the time. A is just the relation between the $MTTF$ and the total time $MTTF + MTTR$.

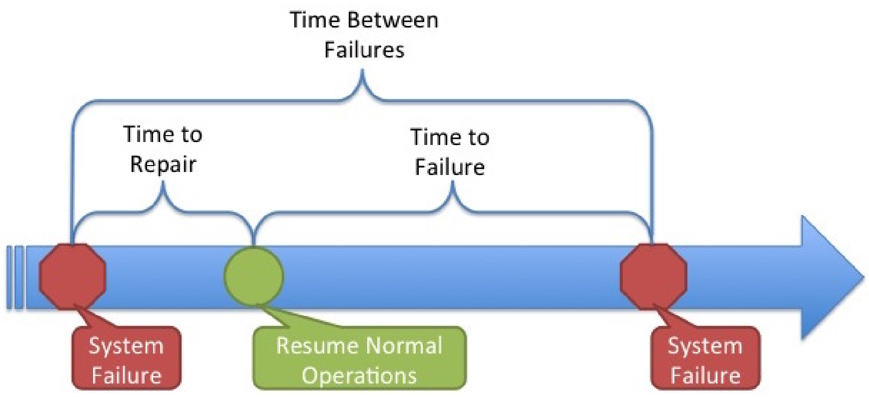


Fig. 3.4: MTTF and MTTR over the time

Memory leaks, unterminated threads, unreleased locks and file fragmentation impact on MTTF. Expected behavior in the presence of these anomalies is that MTTF drecreases as the occurrence of them increases. If a lot of memory leaks occur, they impact on the memory, reducing the free available memory and leading the system to work in a degradation mode until all the memory and the swap space will be filled. Of course, a failure will occur earlier than memory leaks did not occur, then MTTF decreases respect to an ideal behavior (i.e. without anomalies). Equation 3.1 shows how decreasing MTTF, the system availability A decrease too. It means that focussing the attention on the MTTF trend, there is a first overview of what is happening inside a system, revealing the presence of possible anomalies. As explained in [1], even if a software design cares about software aging, it is impossible to avoid totally above mentioned anomalies effects, so the natural consequence was that software and system design thought ways to face them in proactive and reactive manner.

3.2 Management

Following the definition of *Availability*, a management technique should be able either to increase the MTTF or to reduce the MTTR. Generally to reduce the MTTR, some recovery techniques are exploited organizing them on different levels[3]. Most of the bugs are quickly fixed by faster recovery techniques while more serious bugs are delegated to slower but more effective recovery techniques. All these techniques can act in two main ways:

- **Reactive approach:** when a failure is detected then a manage routine starts. The maintenance starts only if a failure will be detected otherwise the system continues to run even if in degradation mode.
- **Proactive approach:** this approach is smarter then the previous one. Collecting some useful infos by the system, proactive management techniques are able to execute a recovery before the system crashes, allowing to establish some fixing points of interest. It should be when the workload is below a particular threshold while an observed feature is above a critical threshold (e.g. arrival rate in web services is below a particular threshold and the used swap space is above a critical threshold).

The main reactive management technique is the replication of software. It is useful in non – aging related Mandelbugs, because of their unpredictable nature so they cannot be anticipated and must be react to. To face this bugs it is important to design management routines acting as fast as possible. On the other hand, aging – related bugs are such that they can cause a predictable increasing failure rate and degraded performance while the system is up and running. This is the case in which a proactive management technique is preferred to the reactive one.

The most important and well known proactive management technique is the *Software Rejuvenation*[4]. Proactive Software Rejuvenation cleaning the system internal state and resetting the system runtime may effectively reduce the failure rate and improve performance. It can be performed rebooting some modules, processes or, in the worst case, the entire system too. A drawback is that, even if it can be performed at different granularities, costs are incurred in terms of scheduled downtime for at least some part of the system.

3.2.1 Time based Software Rejuvenation

Before the system is deployed, there is a phase for collecting parameters, based on the past experience. Infos about time to failure have to be got due to aging related failures. During the system run, time to failure data are collected and used to parameterize the model. Using them, a software rejuvenation can be executed at "fixed time", adapting these fixing points to observed time to failure data, optimizing the schedule. In this case, actions are performed independently of the actual working state of the system.

3.2.2 Load based Software Rejuvenation

This kind of Software Rejuvenation does not need time to failure inputs. It is performed just monitoring system resources at run-time and predicting the time to exahustion of resources. Predict a single resource exahustion was trivial, the challenge was to predict a system failure based on a complex combination of more resources. On the other hand, benefits of proactive management can be significant, in terms of both system service downtime and system performance.

More recent works, show that proactive management of software anomalies can efficiently exploit Machine Learning (ML) algorithms to predict the time to crash of applications. The system is trained until crashing in the presence of anomalies, and some system feature have been collected. After that, values of system features are fed to a ML algorithm for building a model to predict the *Remaining Time To Crash (RTTC)* of the system[21]. The benefits in using this approach are that a recovery action can be performed before the predicted failure time or even before that the system performance goes below a given level. Again, in [6] is showed that using ML-based approach can be predicted the occurrence of both system crashes and other events (e.g. violations of performance thresholds), also in the presence of different kinds of software anomalies.

Chapter 4

PCAM Framework

In this section a ML-based framework will be presented for Proactive Client-server Application Management (PCAM)[5] in the cloud. In a nutshell, the PCAM framework is able to predict the remaining time to the occurrence of some unexpected event of a virtual machine hosting a server instance, through optimized ML-based models and continually measuring system features. In particular, PCAM exploits models produced by the ML framework presented in [6], called F^{2}PM.

The framework works with replicated server instances, assuming that they are deployed on virtual machines (VMs) provided by a cloud IaaS (Infrastructure as a Service). The instances can be added or removed at run-time to dinamically scale the server pool according to the system workload. Of course, anomalies can affect VMs execution, lead them to fail or to work in degradation mode over the time. PCAM is able to detect these situations then to act in recovering VM on the basis of the run-time predictions of its *Remaining Time To Failure (RTTF)* and on monitored VM performances. The RTTF is the time when a failure condition is expected to be true. The failure condition stays for a crash or for a violation of user-defined performance thresholds. Using PCAM allows to work on complex common application deployment, with replicated server instances on different machines where different kinds of anomalies can occur.

The goal of this framework is to improve the system availability, overcoming the drawback shown in Section 3.1.3, and the system performance. It has the advantage to be not tied to specific applications because it has just to retrieve system parameters at hosting machines and operating system level, so it acts in a completely *application-agnostic way.*

Exploiting the ideas in [21], PCAM extends the set of possible ML algorithms and, because of its reliance on F^{2}PM, it can face both a differentiated set of anomalies and user-defined rules to identify failure point. Again, now the framework is able to work on a set of distributed VMs in the cloud and not only with a couple of VMs.

A client-server application model is considered. Clients requests are served by a set of replicated servers running on VMs of a cloud IaaS. PCAM acts when a VM is detected to approach the failure condition. It uses sofwtare rejuvenation to bring the VM in a clean state. Software rejuvenation is performed simply rebooting the machine, but the framework provides the possibility to set other custom techniques following the user will.

4.1 Framework architecture

The PCAM architecture is designed to have a VM acting as a controller, called VMC. A k couples of VMs act as replicated servers. Inside a couple, a VM is taken as *active* anche the other one is taken as *stand-by.*

VMC needs the following components:

- **Communication Unit (CU):** it has to communicate with all the replicated VMs.
- **Prediction Unit (PU):** it returns the computed RTTF of a VM.
- **Load Balancing Unit (LBU):** it forwards remote clients requests to active replicated VMs.
- **Managing Unit (MU):** it manages all the VMs. It processes the incoming VMs messages and decides when a VM has to be rejuvenated.

On the VMs composing the replicated servers, the following components are installed:

- **Communication Unit (CU):** it has to communicate with the VMC.
- **Measurement Unit (MeU):** it collects the local value for a VM's system features.
- **Local Managing Unit (LMU):** it sends collected features to VMC and waits for commands from the VMC to start the VM rejuvenation.

To better understand how this architecture appears, the following Figure 4.1 is provided.

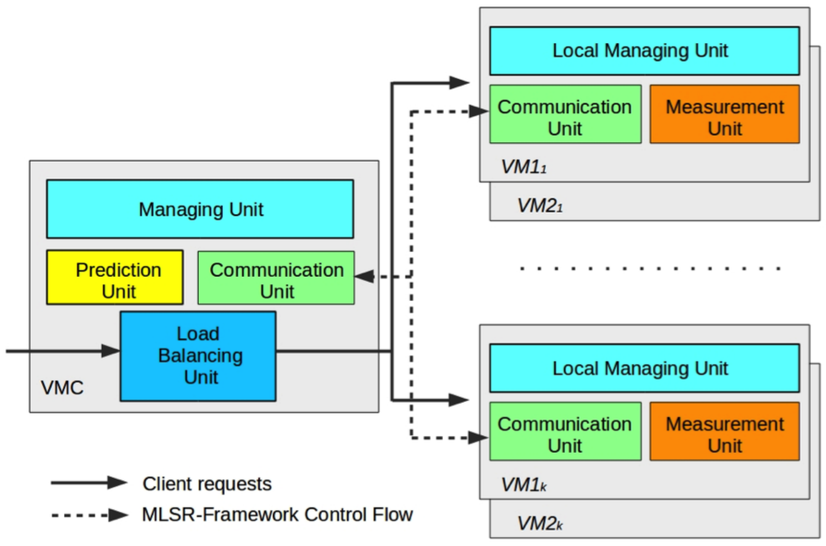


Fig. 4.1: PCAM architecture

MU knows the couples of VMs. At the start up, the MU activates one VM for each couple marking it as *active*, then marking *stand-by* the other one. Remote client requests are forwarded only to the active VMs passing through LBU. In the Figure 4.1, dotted lines represent infos exchanged among VMs and VMC. VMC is able to implement an *On-line control loop* receiving values about the VM features and sending it the rejuvenation command.

As already seen, PU provides RTTF as a response to a query executed by the MU. The prediction unit has to rely on ML-based model to compute this result. In the following section the *Framework for building Failure Prediction Models* (F^2PM) used by PCAM, will be presented.

4.2 F^{2}PM

The F^{2}PM framework is able to build up prediction models starting from a dataset of system features collected on VMs that are running in the presence of anomalies[6]. This framework exploits the client-server paradigm. In particular the *Feature Monitor Client (FMC)* and the *Feature Monitor Server (FMS)* are respectively installed on the client VM (to collect the system features) and on the VM that acts as a server. FMS has to collect continuosly system features incoming from the monitored VMs (e.g. client VMs). In particular, when a VM reach a failure condition, the failure event will be registered and the VM will be restarted. A database will contain all the collected data over the time. One of the main feature of F^{2}PM is the possibility to choose a subset of monitored system features having stronger impact on the RTTF prediction. Using Lasso Regularization[22], increasing the λ parameter, the amount of data to be sent from LMUs of VMs to MU of VMC can be significantly reduced, in order to reduce the training time.

In addition to the selected features, other infos are considered, such as the slope[6]. It represents a simple approximation of a derivative function, catching the dynamics of the system relating to the selected features. Formally:

$$slope_j = \frac{x_j^{end} - x_j^{start}}{n} \tag{4.1}$$

where x_j^{end} and x_j^{start} represents respectively the value of the feature j of the first and the last original datapoint falling in the time interval. Thanks to the slope, the evolution of a particular resource usage on VMs can be captured more accurately, again, the impact of the monitored resource can be related to the RTTF trend on the VMs.

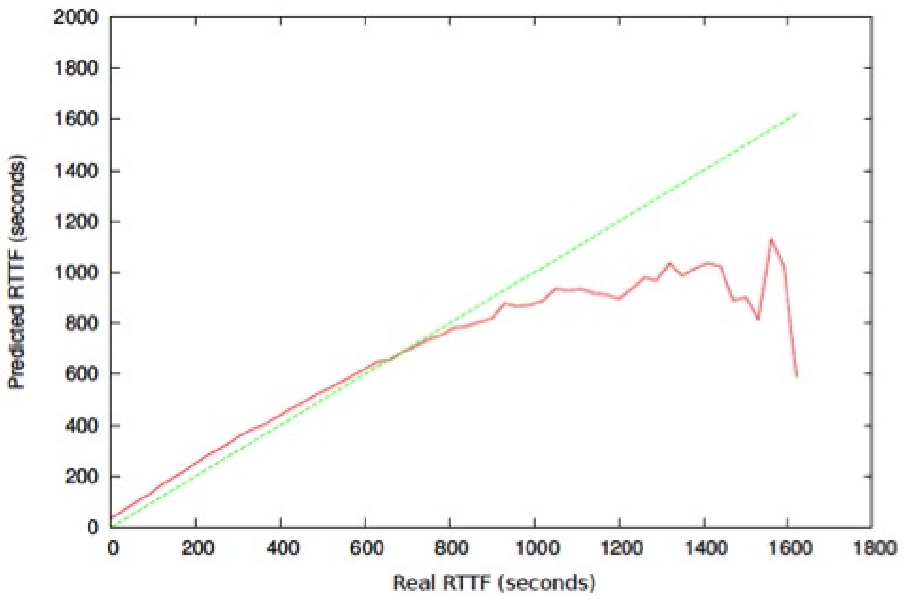


Fig. 4.2: RTTF prediction accuracy using MP5 algorithm

Finally the dataset is used by WEKA[23] to generate the RTTF prediction models exploiting several ML techniques like: Linear Regression, M5P, REP-Tree, Lasso as a Predictor, Support-Vector Machine (SVM), and Least-Square Support-Vector Machine[6]. The Figure 4.2 shows the accuracy in predicting the RTTF using the MP5 algorithm. The real RTTF is represented by the green line while the predicted RTTF is represented by the red line. Again, x-axis shows the real RTTF values while the y-axis shows the predicted RTTF values. The plot shows how, close to the failure point (at time 0), the prediction becomes more accurate.

Finally, once the RTTF prediction model is generated, F^{2}PM provides some indicators of each model, used by PCAM to allow the user to select the most accurate model to compute the RTTF estimation. The selected model will be used at run-time by PU in PCAM framework. The framework allows the user to select interested features to be monitored, in this context, the following features are considered:

- n_{th} is the number of active threads in the system.
- M_{used} is the amount of memory used by applications.
- M_{free} is the amount of free memory in the system.
- M_{shared} is the amount of used memory in buffers shared by applications.
- M_{buff} is the amount of memory used by the underlying OS to buffer data.
- M_{cached} is the amount of memory used for caching data.
- SW_{used} is the amount of used swap space.
- SW_{free} is the amount of free swap space.
- CPU_{user} is the percentage of CPU time spent by normal processes executing in user mode.
- CPU_{ni} is the percentage of CPU time spent by high priority processes executing in user mode.
- CPU_{sys} is the percentage of CPU time spent by processes executing in kernel mode.
- CPU_{iow} is the percentage of CPU time spent by processes waiting for I/O to complete.
- CPU_{st} is the percentage of CPU time spent by processes waiting for services of other processes.
- CPU_{id} is the percentage of CPU idle time.

Of course, PCAM use the same system features to build the prediction models.

Just for the sake of clearness, Figure 4.3 shows the entire F^{2}PM architecture and a short description of it will be provided.

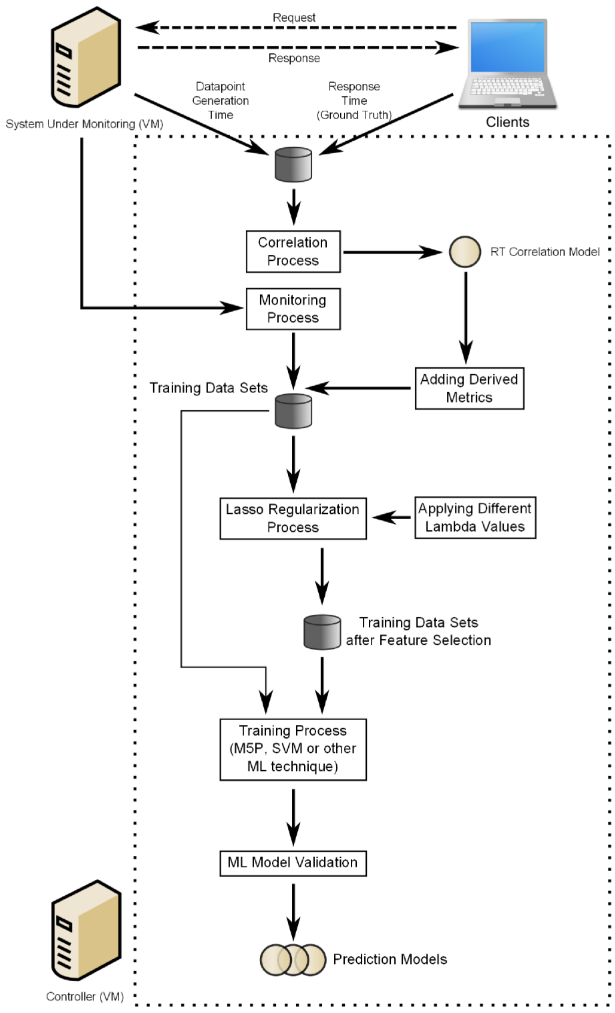

Request
Response
Datapoint Generation Time
Response Time (Ground Truth)
System Under Monitoring (VM)
Clients
Correlation Process
RT Correlation Model
Monitoring Process
Training Data Sets
Adding Derived Metrics
Lasso Regularization Process
Applying Different Lambda Values
Training Data Sets after Feature Selection
Training Process (M5P, SVM or other ML technique)
ML Model Validation
Prediction Models
Controller (VM)

Fig. 4.3: F^{2}PM architecture

From the top, aggregated datapoints are generated on the basis of user-defined time interval. A more detailed description of this phase can be found in [6], it is irrelevant for the purpose of this work. More interesting is the management of the *Training Data Set*. As already seen, a user may set a λ value to create a new training data set as a subset of all the monitored system features. In both cases (e.g. applying or not Lasso Regularization), a *Training Process* is applyed to the data set, towards an ML model validation. Finally the architecture provides the prediction models used by PCAM to estimate the RTTF on the VMs.

4.3 On-line Control Loop

The final step for PCAM is to perform the on-line control loop. It consists in a certain number of steps in order to switch machines from *active* to *stand-by* (and viceversa) when the VM predicted RTTF is below a threshold T. In particular PCAM executes the following steps:

1. The LMU of an active VM has to collect the local measurements of the system features and then send them to VMC.
2. Once MU on VMC has received the measurements from the LMU, it queries the PU using them, getting the predicted RTTF. Only if the predicted RTTF is smaller then a threshold T, the MU performs the following actions:
   (a) Each MU has a list of VMs. In this list it marks the monitored VM as *stand-by* and another one as *active*.
   (b) It communicates to the LBU the new list of active VMs.
   (c) It sends to the just switched in stand-by VM the *rejuvenate* command to start the rejuvenation procedure.
   (d) Finally, it starts to receive measurements from the new activated VM.

When a VM receives the *rejuvenate* command, it does not start immediately the rejuvenation procedure but it has some extra time to finish pending request. The MU sends the rejuvenation command only after it switched the VMs state, to avoid the case in which incoming client requests can be lost. In this way new incoming requests are forwarded directly to the new activated VM while the other VM manage pending requests before to execute the rejuvenation procedure.

The Figure 4.4 illustrates how is the rejuvenation for a couple of VMs.

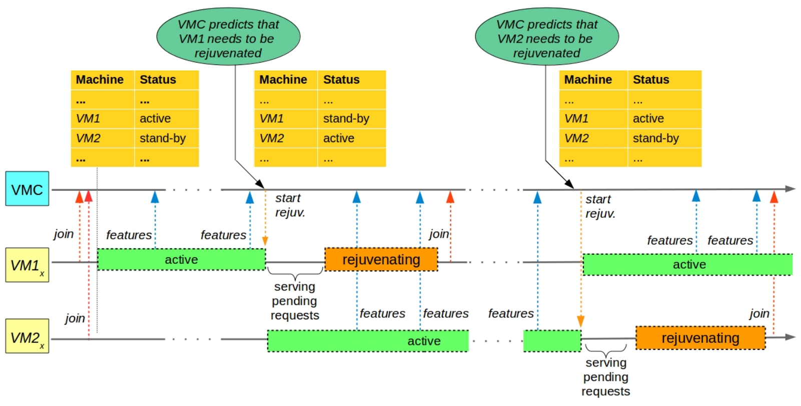


Fig. 4.4: Rejuvenation policy for a couple of VMs

The above mentioned threshold T can be tunable by the user. It simply represent a time value used to manage the rejuvenation procedure. In particular this *safety value* is used by PCAM to determine the time instant when a VM has to be rejuvenated before the predicted failure time. It is exactly the time in which a VM can execute pending requests. T should be set in the same magnitude order of the request response time and it would be used to reduce the effect of MTTF prediction error on the system availability. If the value of T is low, a VM is rejuvenated a few time before the predicted failure time, then even a very small overstimation of MTTF could prevent

PCAM from rejuvenating a VM before the actual failure time. For high value of T, a VM may be rejuvenated early it can continue to run in acceptable conditions.

Chapter 5

Proactive Workload Management in Multi-Cloud Environment with PCAM

In this section, a proactive workload management in multi-cloud environment will be presented, exploiting the PCAM framework presented in Chapter 4, in the presence of software aging. As already seen in Section 3.2, one of the most well-known technique to overcome the issues related to the accumulation of anomalies is the *Software Rejuvenation*, in particular, the *Proactive Rejuvenation* allows to preventively force the application or hosting system to a clean state before a time when a crash is predicted to occur (i.e. a state where the system/application is known to work whithout the presence - or with reduced number - of anomalies). Results in [6] and [5] have shown how proactive software rejuvenation allow to increase the availability and reduce the response time of virtualized (web) applications. This work considers a more complicated multi-cloud scenario, where the same application is replicated in differentiated cloud regions (even at a geographical scale), possibly managed by different providers, and possibly organized as a hybrid cloud infrastructure. In this context it was considered a geographically distributed hybrid cloud, composed of VMs hosted on Amazon EC2 service (in Ireland and in Frankfurt), and VMs hosted on a private server in Munich.

A more detailed architecture will be presented in Section 5.2.

5.1 Goals

This work want to build up an innovative framework to increase the availability of the application and to reduce the response time seen by the users, at the same time. Being in a distributed environment, this framework addresses the following scenarios: on the one hand, the management of VMs in a single cloud region is performed using ML-based prediction models, which allow to act software rejuvenation in case of the accumulation of anomalies, on a single VM scale; on the other hand, the framework is able to redirect requests by remote users to different cloud regions so that (on a global basis) the response time is kept below a given threshold, and the rejuvenation rate of the VMs is minimized.

The application is replicated on any number of VMs whose state and execution behavior is managed by a set of distributed controllers (hosted in different virtual machines), which are in charge of controlling the effects of accumulation of anomalies. Each controller has to make an online prediction of the RTTF using ML algorithms of any managed VM. The RTTF prediction models allow to the system administrator to select a set of rules to establish when, during the training phase, an ML algorithm can consider a VM as failed.

Overall, the framework allows an easy deploy and management of a replicated client/server (web) application on a hybrid cloud. In particular it is easy to activate new instances (e.g. when the workload increases from remote users), and trigger the rejuvenation process to contrast the effects of software aging on each VM.

Finally, the main goal of the distributed application is to allow to transparently redirect user incoming connections towards any region, depending on the current load and state of accumulation of anomalies, in all the cloud regions by relying on MTTF value.

5.1.1 Load Balancing technique based on region MTTF

The controller of the cloud region monitors the MTTF of each VM into its own cloud segment. The MTTF is used to determine when additional VMs should be activated to serve incoming requests. The MTTF can be seen as a value telling if the current computing power, that is the set of VMs working within the region, is sufficient to manage the current workload, and if the generation rate of anomalies determines a too frequent rejuvenation of active VMs. The idea is that adding new VMs can locally distribute the incoming workload then reduce the rejuvenation rate.

In this way the framework is able to simulate most common client/server applications, in which a client can decide to connect to a specific region either by a-priori knowing the IP addresse of regions' entry points (e.g. a list of IP addresses is cabled into the client's code), or by retrieving the IP of any region by relying on an additional directory service (e.g. public DNS). Most of cases one cloud region may receive more requests than other, affecting a decrease in the average MTTF of the region, that is the average MTTF of all VMs in the region. The controllers of different regions share this knowledge, and by exploiting a distributed algorithm, they decide upon a shared policy to redirect local requests to remote regions. In turn, this levels the MTTF of all regions, and therefore prevents the activation of additional VMs in overloaded ones. Thus, the framework is able as well to reduce the cost associated with the activation of unnecessary additional VMs, exploiting the already-available computer power located in geographically different cloud regions.

5.2 System Architecture

As mentioned, the cloud resources can be distributed over the services offered by different cloud providers, and/or over services offered by the same cloud provider in different geographically locations, and/or over private cloud infrastructures. The VMs hosted in a given region by one single cloud provider is referred as a *cloud region*. Each cloud region has to contains at least the following set of VMs:

- **Controller (CON):** it is the VM that has to supervise the execution of the VMs in the cloud region.
- **Load Balancer (LB):** The Load Balancer is the VM that receives remote user connections to redirect them either to specific VMs in the cloud region or towards remote cloud segments. It represents the entry point of the distributed system from the outside world and it receives from CON information about which are the currently available VMs.
- **Computing nodes (CN):** they represents the VMs that are hosting a copy of the application. As already presented in [5] each of them can be either in the *ACTIVE* state or in th *STAND_BY* state. When a

CN is in the *ACTIVE* state, it is processing requests form the remote clients, while when it is in *STAND_BY* state it is just waiting for the activation command from CON when the system needs its activation. Switching to the *ACTIVE* state, it begins to process incoming requests. The activation command is received by the VM in *STAND_BY* state when a currently *ACTIVE* VM has to be rejuvenated or whenever CON decides that a higher number of VMs are necessary to manage the incoming requests, given the current load of the system.

The whole system is composed of any number of cloud regions. The framework does not mind if they are hosted by both public providers and private infrastructures, making it viable in the case of public, private and hybrid cloud environments. Again, it is important that they can be geographically distributed: the system is resilient as well to partitioning or disasters.

Figure 5.1 shows the system architecture. Each cloud region is represented from a cloud (i.e the set of CNs), a LB and a CON. They are connected by an overlay network, and the clients can communicate directely only with the closest LB, the entry point for the distributed application.

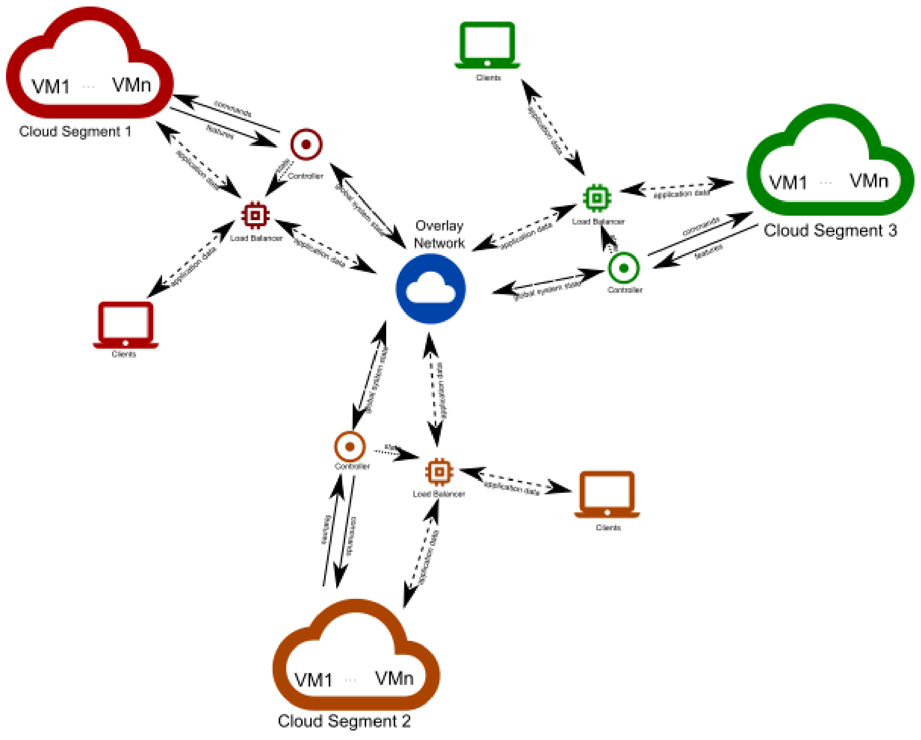


Fig. 5.1: System architecture

Last step before the description of the Load Balancing strategy in the system, the overlay network features will be descibed. In particular, to allow the exchange of knowledge among the controllers, they must rely on reliable communication system, and due to the fact that remote clients' requests are redirected to other VMs hosted in other cloud regions, the framework must ensure that redirecting a request does not affect its response time. The overlay network is designed to allow communication among different cloud regions. Each of them has to run two proxy agents (one transmission agent and one reception agent) handling overlay traffic of all virtual machines of its cloud. The agents are hosted into the LBs, that are in the middle between remote users and the CNs in the system, and act as "connection proxies" among all the different LBs. The agents' purpose is to constantly monitor the information used to determine the optimal path in the network to each possible destination, that is all the other LBs in the distributed environment. Whenever a transmission agent is requested to deliver a packet to any LB in the system, the information is sent to the destination agent following the optimal path. Finally, thanks to the overlay network, each CON-CON communication is directed towards it. It is sure that the communication among controllers is guaranteed even in the case of network partitions. Again, the remote client requests are redirected to other regions with the minimal latency offered by the current state of the network.

5.3 Load Balancing strategy

The core underneath the proactive workload management in multi-cloud environment is the load balancing strategy. In this section it will be described how it works in order to provide a clear knowledge about the relation between all the components composing a cloud region and the logical steps executed by the framework.

First of all, the controllers have to take coherent decision. To do that, a *leader election* algorithm is required, in order to elect, among all the CON nodes, one leader. The presence of a leader is necessary, indeed it is the only one that can collect usage data from other controllers and can decide what is the best (global) policy to keep the MTTF of the system above a certain threshold. Keeping the MTTF greater than a critical threshold it allows to keep the response time below a given threshold and the rejuvenation rate of all the VMs into the distributed application can be minimized.

In this context, it was chosen the leader election algorithm presented in [7]. It is an efficient algorithm which can scale to any number of participants (thus, ensuring scalability of the framework to any number of cloud regions) and which is highly resilient to changes in the network of participants. Specifically, by relying on this algorithm, the framework is able to enforce dynamic network reconfiguration, even in case of multiple node and link failures in high-speed networks with arbitrary topology. In particular, CON nodes keep routing tables which allow to store information about the participating nodes (that is, the other controllers) and they asynchronously update their content, namely these routing table are incrementally generated/updated during the lifetime of the distributed application.

Essentially, each CON uses its public IP as its ID number. When the application starts (or whenever controllers suspect the current leader to be failed) each CON node must exchange information about its ID only with neighbour CONs, in order to determine what is the (non-failed) CON node with the highest ID. This is the node which is distribudetly elected as the leader.

The final goal of the load balancing strategy, is to distributed the (new) incoming connections from remote clients of the (general) application hosted by the virtual cloud infrastructure to the cloud regions which are currently less loaded.

Let's start analyzing one single cloud region, just for the sake of simplicity. Each cloud region has several *ACTIVE* VMs and others in the *STAND_BY* state. The ML-based prediction models presented in [6] can be used to etimate the (current) RTTC. The predicted values are generated using both measured features (i.e. the current state of some measures of interest, such as used memory or user CPU usage) and derived metrics. In particular, in this context, the derived metrics are used differently from the actual prediction way in which they are used to compute RTTF by the controllers. The slopes represent the effect of the current clients' request rate λ, due to the fact that derived metrics represent how far the system is approaching the rejuvenation point given both the current workload and the anomaly accumulation rate of a given (set of) virtual machines.

To estimate the capacity of the regions, the framework exploits the ML models generated during the learning phase according to the results in [6]. These above mentioned models are used by the controllers to perform the proactive rejuvenation in VMs, in a given cloud region, according to result in [5]. Given a prediction model, to compute the estimated capacity, the current slopes are passed as inputs beside the measured system features after the rejuvenation of a VM. Thus, the ML-based prediction model returns how much time is estimated for a VM to reach the rejuvenation point from the beginning of its (current) execution. By using the prediction model in this way, the *Mean Time To Failure* (MTTF) of a given VM is provided. By supposing that there are V currently-active VMs, the same evaluation is performed for each $v \in V$, using different prediction models too. Once the framework knows the MTTF value, it is able to compute the *rejuvenation rate* for that particular machine, as shown in Equation 5.1.

$$f_R^v = \frac{1}{MTTF_v} \tag{5.1}$$

More interesting is the *average rejuvenation rate* f_R^s for a single cloud region $s \in S$:

$$f_R^s = \frac{\sum_{v \in V} \frac{1}{MTTF_v}}{|V|} \tag{5.2}$$

Each controller in the distributed system computes its own region $MTTF_i$, then each LB has to communicate to its CON the current clients' arrival rate λ_s for that region. Finally the controller is able to send to the current leader the couple $< f_R^s, \lambda_s >$, then the current leader can compute the *global average rejuvenation rate* using the all the couples received by the other controllers, as shown in Equation 5.3

$$f_R = \frac{\sum_{s \in S} f_R^s}{|S|} \tag{5.3}$$

and the *global clients' arrival rate* as shown in

$$\lambda = \frac{\sum_{s \in S} \lambda_s}{|S|} \tag{5.4}$$

All the just computed values are necessary in the load balancing strategy to compute the probability according to which each incoming connection from any client is redirected towards a given cloud region i. In particular the current leader computes the probabilities as shown in the following equation:

$$p_i = \frac{\frac{1}{f_R^s}}{\sum_{s \in S} \frac{1}{f_R^s}} \tag{5.5}$$

Now that all the values are computed, the leader is able to calculate the *estimated capacity* of a given region in the following way:

$$c_i = \lambda_i \cdot p_i \tag{5.6}$$

Controllers communicate with the leader every T seconds. Only after the leader has collected the values from all the currently active controllers, can make a decision on what is the amount of incoming connections λ_i that each load balancer shoul redirect towards other regions.

Practically, the leader election computes the above mentioned amount solving a fractional multi-knapsack algorithm (F-MKP). The algorithm uses a set of n items and a set of m knapsacks (where $m \leq n$), furthermore:

- p_j is the profit ot item j.
- w_j is the weight of item j.
- c_i is the capacity of knapsack i.

F-MPK selects m disjoint subsets of items such that the profit of the selected items is maximum and each subset can be putted in a knapsack having at least the capacity equal to the total weight ot items in the subset. It can be formally represented as follows:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_{ij} \le c_i \quad , i \in M = 1, \dots, m \\ & \sum_{i=1}^{m} x_{ij} \le 1 \qquad , j \in N = 1, \dots, n \\ & 0 \le x_{ij} \le 1 \end{aligned} \tag{5.7}$$

The portion of the item j that is assigned to the i-th knapsack is referred in this linear problem as x_{ij}. In this framework, i-th load balancer serving the i-th region is mapped to a knapsack with c_i representing the region capacity. The items are the remote user connections and for each item j the weight is equal to cost ($w_i = p_i$). The multi-knapsack problem is hard to solve in relation to resource usage, then the Dantzig's approximation of knapsack is exploited to overcome this issue. It guarantees to keep the error in the approximation result below a 30% threshold.

When the leader computed the fraction of the workload to redirect towards each cloud region, it has to communicate these values to all other participating controllers. Now, each region controller, must communicate to load balancer this information, that keeps the just received results in a *flow* *control matrix*. This matrix respresents the global view of the forwarding probabilities for each load balancer in the system. The algorithm tries to forward all the incoming user requests to the local VMs, if it is not enough to serve all the requests, with a certain probability the remaining connections are forwarded to remote cloud segments. A more detailed description will be provided in the Section 5.4.1.

It is clear that the activation period T according to which the controllers exchange their workload data can be critical. In case of very dynamic workload, a too large value for T would not allow the controllers to capture it, while on the other hand a very small value for T in case of a slowly-changing workload would be a waste of network and computing resources. Therefore, to autonomically capture this, it is used Hill Climbing[24], organized in a set of operational states, which are referred to the operation applied to T, namely *multiplication*, *division*, and *no-operation*. In general, hill climbing is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found.

Each time that the leader has computed the fractions of workload to be redirected, he tries to increase/decrease the value of T by multiplying/dividing it by a number in the interval [1,2]. The leader then sends back to all controllers the new value *T'*, and all controllers install it, so that they are aligned to the new activation period.

The leader keeps track of T, *T'*, the operation applied to T (multiplication or division) and the fraction of workload to be redirected. At the next round, when the leader recomputes again the workload redirection fractions, the leader can check whether the change from *T* to *T'* has produced some effect. In particular, if the applied operation was a multiplication and the fractions have not changed much (namely, the difference is $\leq 10\%$), then the new value *T'* is accepted and the algorithm is iterated. Rather, if the fractions have changed too much, it means that we are facing a change in the workload. Therefore *T'* is discarded, and the operation to be applied is switched from multiplication to no-operation.

On the other hand, in case the applied operation was a division, the checks made by the algorithm are the other way round. In particular, if the fractions change much (i.e their difference is $\geq 10\%$), it means that we are facing a workload change and it is meaningful to continue shrinking the interval *T*. Therefore, *T'* is accepted and the algorithm iterates. On the other hand, if the variation is not substantial, *T'* is descarded and it is switched back to no-operation.

When in the no-operation state, the system has found a value of *T* which is good for the current dynamics. In this case, the leader does not modify *T* (namely, is applies the no-operation) until the variation in the fractions is higher than 10%. In that case, the system starts applying multiplication/division depending in the state from which it came to the *no-operation* one. Specifically, if before reaching the *no-operation* state the algorithm was applying multiplication, then it leaves the *no-operation* state reaching the division one, and vice versa.

With an ϵ probabiity, the algorithm can jump from the *multiplication* to the *division* and vice versa, even if the above described conditions are not met.

There are certain circumstances when simply redirecting the load is not sufficient to ensure availability of a system. As an extreme example, a peak of connections might increase the rejuvenation rate of virtual machines in one or multiple regions to an extent so high that the whole systems become unavailable, simply due to the high number of rejuvenations.

As a countermeasure to this issue, each region's controller monitor the MTTF values shown in Equation 5.2. If this value becomes less than a given threshold (which can be specified at configuration time), then the controller decides to activate new spare VMs. Specifically, this can be easily done by sending a control message to a VM in the pool which is currently in the *STAND_BY* state, so that this VM is brought to the *ACTIVE* state even if no rejuvenation action is taking palce.

It can be noted that if the number of spare VMs is low, this solution can be complemented with services offered by cloud service providers, which allow to create and start new VMs from the system image. Newly instantiated VMs in this framework immediately connect to the region's controller to notify their presence, and they join the pool in the *STAND_BY* state.

In Chapter 6 it will be showed the experimental evaluation of the presented proactive workload management to support the choice of the load balancing strategy just mentioned.

5.4 Components characterization

In this section, the main parts of the implementation of all components will be presented. It helps to better understand how the components communicate among them and how the load balancing strategy can be spread in the cloud. The implementation is executed in ANSI C language, on unix based OS. It allows to monitor system features directly retrieving their values from the system's files (e.g. /proc/meminfo, /proc/cpuinfo,...). There is a transition period in which the system is reaching a full-speed state, where all the components are working and they are communicating among them. The controller is the first component that must be launched to wait the connection from its own balancer. Once they are communicating, the system is able to accept the VMs dedicated to serve user requests. More detailed description will be provided for each component.

5.4.1 Controller

The controller is the component that has to supervise the execution of the VMs in the cloud region. Again, the controller may be either a *Leader* or not. If it is, it means that each other *not-Leader* controller must send infos about its own region (i.e. region arrival rate and region mttf) to it, in order to allow the computation of the global system features by the leader. Using these informations, the leader is able to computes the forwarding probability for each load balancer in the distributed system, then it communicates to each of them the respective probability. It follows a step-by-step description of the most interesting parts in the controller's implementation.

Once a controller is started, first of all, it has to wait for its own load balancer connection. Load Balancer has to establish two different connections with the controller: the first one for exchanging infos about of the VMs' status in its cloud region and the other one for infos about the arrival rate λ. Only after this preliminary logical connection, a new thread is created to manage infos incoming from the load balancer.

Listing 5.1 reports the code:

```c
//First connection
if ((accept_load_balancer(sockfd_balancer, pthread_custom_attr, &
    sockfd_balancer)) < 0)
  exit(1);

if (sock_read(sockfd_balancer, my_balancer_ip, 16) < 0) {
  perror("Error in reading balancer public ip address: ");
}
printf("Balancer correctely connected with public ip address %s\n",
    my_balancer_ip);

//Second connection
if ((accept_load_balancer(sockfd_balancer_arrival_rate, pthread_custom_attr,
     &sockfd_balancer_arrival_rate)) < 0)
  exit(1);

//Start new thread for arrival rate infos
pthread_attr_init(&pthread_custom_attr);
pthread_create(&tid_balancer_arrival_rate, &pthread_custom_attr,
    get_region_features, (void *) (long) sockfd_balancer_arrival_rate);
```

Listing 5.1: Load Balancer connections

Inside the dedicated thread, the controller waits for infos from its own load balancer. In particular when it receives the arrival rate, it is able to compute the regional MTTF. Due to the fact that the controller may be either the leader or not, if it is not, it has to send the couple $< MTTF, \lambda >$ to the leader, then it has to wait for the forwarding probabilities from the leader. If it is the leader, it uses the special position 0 in a global data structure, to store itw own features, then it computes the global arrival rate and the global MTTF. Finally, in both the cases, it can communicate computed probabilities to its own load balancer.

The code is the following:

```c
void * get_region_features(){
...
strcpy(region.ip_balancer, my_balancer_ip);
while (1) {
  if (sock_read(sockfd, &arrival_rate, sizeof(float)) < 0)
    perror("Error in reading in arrival_rate_thread: ");
  pthread_mutex_lock(&mutex);
  compute_region_mttf();
  ...
```

Listing 5.2: Receiving arrival rate from LB

```c
  ...
  //If i am not the leader, send my features to the leader
  if (!i_am_leader) {
    memset(regions, 0, NUMBER_REGIONS * sizeof(struct _region));
    get_my_own_ip();
    strcpy(region.ip_controller, my_own_ip);
    region.region_features.arrival_rate = arrival_rate;
    region.region_features.mttf = region_mttf;
    //Send infos to the leader
    if (sock_write(socket_controller_communication, &region,
        sizeof(struct _region)) < 0) {
      perror("Error in sending region features to leader: ");
    }
    //Waiting for the probabilities
    if (sock_read(socket_controller_communication, &regions,
    NUMBER_REGIONS * sizeof(struct _region)) < 0) {
      perror("Error in receiving probabilities from leader: ");
    }
    //Send probabilities to my own load balancer
    if (sock_write(sockfd, &regions,
        sizeof(struct _region) * NUMBER_REGIONS) < 0) {
      perror("Error in sending regions to my own load balancer: ");
    }
  }
  ...
```

Listing 5.3: Sending features if the controller is not the leader

```c
  ...
  //If i am leader, update my own values
  else {
    //Store infos in the special position 0
    regions[0].region_features.arrival_rate = arrival_rate;
    regions[0].region_features.mttf = region_mttf;
    strcpy(regions[0].ip_balancer, my_balancer_ip);
    update_region_workload_distribution();
    int i;
    //Use the global matrix to retrieve my own load balancer forwarding
    probabilities
    for (i = 0; i < NUMBER_REGIONS; i++) {
      regions[i].probability = global_flow_matrix[0][i];
    }
    //Send probabilities to my own load balancer
    if (sock_write(sockfd, &regions,
        sizeof(struct _region) * NUMBER_REGIONS) < 0) {
      perror("Error in sending regions to my own load balancer: ");
    }
  }
  pthread_mutex_unlock(&mutex);
}
```

Listing 5.4: Sending features if the controller is the leader

For the sake of completeness, it follows the code to compute the regional MTTF and the code to compute the global MTTF and the probabilities:

```c
void compute_region_mttf() {
  ...
  for (index = 0; index < vm_list_size(vm_list); index++) {
    struct virtual_machine *vm = get_vm_by_position(index, vm_list);
    if (vm->state == ACTIVE && vm->mttf > 0) {
      mttfs = mttfs + vm->mttf; number_active_vms++;
    }
  }
  region_mttf = mttfs / number_active_vms;
  ...
}
```

Listing 5.5: Compute the regional MTTF

```
void update_region_workload_distribution() {
  float global_mttf = 0.0;
  float global_arrival_rate = 0.0;
  int index;
  int number_of_regions = 0;
  for (index = 0; index < NUMBER_REGIONS; index++) {
    if (strnlen(regions[index].ip_controller, 16) != 0
        && !isnan(regions[index].region_features.mttf)) {
      global_mttf = global_mttf + regions[index].region_features.mttf;
      global_arrival_rate += regions[index].region_features.arrival_rate;
    }
  }

  for (index = 0; index < NUMBER_REGIONS; index++) {
    if (strnlen(regions[index].ip_controller, 16) != 0) {
      if (isnan(regions[index].region_features.mttf)) {
        regions[index].probability = 0;
      } else if (isinf(regions[index].region_features.mttf)) {
        regions[index].probability = 1;
      } else {
        regions[index].probability = regions[index].region_features.mttf
            / global_mttf;
      }
    }
  }
  ...
```

Listing 5.6: Compute global values

```
  ...
  //Compute the global flow matrix
  float f[NUMBER_REGIONS], p[NUMBER_REGIONS];
  memset(f, 0, sizeof(float) * NUMBER_REGIONS);
  memset(p, 0, sizeof(float) * NUMBER_REGIONS);
  for (index = 0; index < NUMBER_REGIONS; index++) {
    if (global_arrival_rate != 0
        && !isnan(regions[index].region_features.mttf)) {
      f[index] = regions[index].region_features.arrival_rate
          / global_arrival_rate;
      p[index] = regions[index].probability;
    }
  }

  memset(global_flow_matrix, 0,
      sizeof(float) * NUMBER_REGIONS * NUMBER_REGIONS);
  calculate_flow_matrix(global_flow_matrix, f, p, NUMBER_REGIONS);
  printf("------------------\nGlobal Flow Matrix:\n");
  print_matrix(global_flow_matrix, NUMBER_REGIONS);
  printf("------------------\n");

}
```

Listing 5.7: Calling the function to compute global flow matrix

The code represented in Listing 5.7 calls the core function for the load balancing strategy. In the following Listing 5.8 it is provided the whole entire code of the algorithm to compute the global flow matrix (Section 5.3). The matrix represents the source load balancers LB_i^S on the rows, and the destination load balancers LB_i^D on the columns. The cells matrix[i][i] represent the probability that an incoming user request is directely managed in the local cloud region. The cells matrix[i][j] with $i \neq j$, represent the probability that an incoming user request will be forwarded to LB_j^D. The algorithm tries to forward all the incoming user requests to the local VMs, represented by the matrix diagonal. If it is not enough to serve all the requests, with a certain probability the remaining connections are forwarded to remote cloud segments. In particular the leader computes the global flow matrix, then it will send to each load balancer its own probability, and each of the load balancer must follow the received values to forward the incoming traffic.

```
void calculate_flow_matrix(float *M, float *f, float *p, int size) {
  int i=0, j=0, current=0;
  float residual, sum;
  float * f_residual = (float*) malloc(sizeof(float)*size);
  memcpy(f_residual,f,sizeof(float)*size);
  memset(M,0,sizeof(float)*size*size);
  for (i=0; i<size; i++) {
    if (f[i] <= p[i]) {
      M[i*size+i]=f[i];
      f_residual[i]=0;
    } else {
      M[i*size+i]=p[i];
      f_residual[i] = f[i] - p[i];
    }
  }
  for (i=0; i<size; i++) {
    current =0;
    while (f_residual[i]>0) {
      if (i!=current) {
        sum=0;
        for (j=0; j<size; j++) {
          sum+=M[j*size+current];
        }
        residual=p[current]-sum;
        if (residual>f_residual[i]) {
          M[i*size+current]+=f_residual[i];
          f_residual[i]=0;
        } else if (residual>0) {
          M[i*size+current]+=residual;
          f_residual[i] -= residual;
        }
      }
      current++;
    }
  }
  for(i=0; i<size; i++)
    if(f[i] != 0){
      for(j=0; j<size; j++){
        M[i*size+j] /= f[i];
      }
    }
}
```

Listing 5.8: Compute global flow matrix function

The next step in the controller implementation is to manage either the relation with the other controllers (if it is the leader), or with the leader.

The latter case is easy, it has just to prepare the connection with the leader, as showed in the Listing 5.9.

```c
...
//Connect to the leader if i am not the leader
else {
  struct sockaddr_in controller;
  controller.sin_family = AF_INET;
  controller.sin_addr.s_addr = inet_addr(leader_ip);
  controller.sin_port = htons((int) GLOBAL_CONTROLLER_PORT);
  if (connect(socket_controller_communication,
      (struct sockaddr *) &controller, sizeof(controller)) < 0) {
    perror("Error in connection to leader controller: \n");
  }
  ...
}
```

Listing 5.9: Connect to the leader if the controller is not

If the controller is the leader, it must be able to handle the connections incoming from other not-leader controllers. A new thread is dedicated for this purpose, it has to accept the new incoming connections from the other controllers, then it has to launch a new thread to manage the communication.

```c
void * controller_communication_thread(void * v) {
  ...
  while (1) {
    if ((sockfd = accept(socket_controller_communication,
        (struct sockaddr *) &incoming_controller, &addr_len)) < 0) {
      perror("Error in accepting connections from other controllers: ");
    }

    pthread_attr_init(&pthread_custom_attr);
    pthread_create(&tid, &pthread_custom_attr, update_region_features,
        (void *) (long) sockfd);
  }
}
```

Listing 5.10: Accept incoming connections from controllers

When the connection is established between the leader and one other controller, it has to collect the features incoming from the not-leader controller. Again, once the features are received, the leader has to compute the global values, then the global flow matrix and finally it has to communicate the right probabilities (i.e. using the loop index) to the not-leader controller. The Listing 5.11 shows the code, where the function to compute the global flow matrix is the same already showed in the Listing 5.6 and Listing 5.7.

```
void * update_region_features(void * arg) {
  ...
  while (1) {
    if (sock_read(sockfd, &temp, sizeof(struct _region)) < 0) {
      perror(
          "Error in reading from controller in update_region_features: ");
    }
    pthread_mutex_lock(&mutex);
    for (index = 1; index < NUMBER_REGIONS; index++) {
      if (!strcmp(regions[index].ip_controller, temp.ip_controller)
          || (strnlen(regions[index].ip_controller, 16) == 0)) {
        strcpy(regions[index].ip_controller, temp.ip_controller);
        strcpy(regions[index].ip_balancer, temp.ip_balancer);
        regions[index].region_features.arrival_rate =
            temp.region_features.arrival_rate;
        regions[index].region_features.mttf = temp.region_features.mttf;

        update_region_workload_distribution();

        int i;
        for (i = 0; i < NUMBER_REGIONS; i++) {
          regions[i].probability = global_flow_matrix[index][i];
        }
        if (sock_write(sockfd, &regions,
        NUMBER_REGIONS * sizeof(struct _region)) < 0) {
          perror(
              "Error in sending the probabilities to all the other
    controllers: ");
        }
        break;
      }
    }
    pthread_mutex_unlock(&mutex);
  }
}
```

Listing 5.11: Update the region features

After that the backbone network is created to pass all the needed informations, finally the controller can start to accept incoming CNs. A function is executed in an infinite loop to always listen for new incoming CNs' connections.

```
  ...
  //Accept new clients
  while (1) {
    accept_new_client(sockfd, pthread_custom_attr);
  }

  ...
```

Listing 5.12: Accepting new VMs

When a new VM is accepted some preliminary operations have to be executed before start getting the system's features values of the monitored VM. First of all, the thread has to check if the number of active VMs is below a given threshold, then if the check failed, it put the VM in *ACTIVE* state and notify the choice to the load balancer, otherwise it put the VM in *STAND_BY* state. In both the cases the VM has to be added in a global data structure to take trace of the region composition. Only after this important control, a new thread is activated to allow the dedicated communication between the controller and the VM.

```c
void accept_new_client(int sockfd, pthread_attr_t pthread_custom_attr) {
  ...
  // store here infos from CNs
  struct vm_service service;

  struct sockaddr_in client;
  addr_len = sizeof(struct sockaddr_in);

  // get the first pending VM connection request
  if ((socket = accept(sockfd, (struct sockaddr *) &client, &addr_len))
      == -1) {
    perror("accept_new_client - accept");
  }
  ...
  // increment number of connected CNs
  pthread_mutex_lock(&mutex);

  //Store the VM's infos
  struct virtual_machine * new_vm = (struct virtual_machine *) malloc(
      sizeof(struct virtual_machine));
  strcpy(new_vm->ip, inet_ntoa(client.sin_addr));
  new_vm->socket = socket;
  new_vm->port = ntohs(client.sin_port);

  //If a new ACTIVE machine is necessary...
  if (get_number_of_active_vms(vm_list) < number_of_active_vm) {
    strcpy(vm_op.ip, inet_ntoa(client.sin_addr));
    vm_op.port = htons(8080);
    vm_op.service = service.service;
    vm_op.op = ADD;
    send_command_to_load_balancer();
    new_vm->state = ACTIVE;
  } else {
    new_vm->state = STAND_BY;
  }

  add_vm(new_vm, &vm_list);
  // make a new thread for each VMs
  pthread_attr_init(&pthread_custom_attr);
  if (pthread_create(&tid, &pthread_custom_attr, communication_thread,
      (void *) new_vm) != 0) {
```

```
    perror("Error on pthread_create while accepting new client");
  }
  pthread_mutex_unlock(&mutex);
}
```

Listing 5.13: Accepting new VMs function

Finally the main thread starts to collect all the informations to compute the MTTF and RTTF related to a particular active VM. If the VM is in stand-by, the controller will not analyze its features but it will send always the "CONTINUE" command. In particular the controller wait for the features from the VM. Once it has received, it is able to predicte the MTTF and the RTTF. This latter value it is used to decide if a VM has to be rejuvenated or it can continue its job. Of course, it could happen that a connection with a VM will be lost (e.g. reset by peer). In both cases (i.e. rejuvenation procedure, connection lost) some operations has to be executed. The socket will be closed and the controller has to notify to the load balancer that the VM is not more available for the application. The main thread dies when one of these just mentioned cases happen. The following Listing 5.14 reports the main thread code:

```
void * communication_thread(void * v) {

  struct virtual_machine *vm = (struct virtual_machine*) v;
  ...
  while (1) {
    fflush(stdout);
    bzero(recv_buff, BUFSIZE);
    //Collect the features
    if ((numbytes = sock_read(vm->socket, recv_buff, BUFSIZE)) == -1) {
      printf("Failed receiving data from vm %s with sockid %i\n", vm->ip,
          vm->socket);
      perror("sock_read: ");
      if (errno == EWOULDBLOCK || errno == EAGAIN) {
        printf("Timeout on sock_read() while waiting data from VM %s\n",
            vm->ip);
      }
      break;
    } else if (numbytes == 0) {
      printf("vm %s is disconnected\n", vm->ip);
      break;
    }
    //If the VM is in the ACTIVE state
    if (vm->state == ACTIVE) {

      fflush(stdout);
      //Store the features
      get_feature(recv_buff, &current_features);

      // fill init_features only the first time
      if (!flag_init_features) {
        memcpy(&init_features, &current_features,
            sizeof(system_features));
        flag_init_features = 1;
      }
      // at least 2 sets of features needed
      if (vm->last_system_features_stored) {
        float mean_time_to_fail = get_predicted_mttf(ml_model,
            vm->last_features, current_features, init_features);
        vm->mttf = mean_time_to_fail;
        float predicted_time_to_crash = get_predicted_rttc(ml_model,
            vm->last_features, current_features);

        //It RTTC is below the REJ threshold, then rejuvenate
        if (predicted_time_to_crash < (float)TTC_THRESHOLD) {
          vm->state == REJUVENATING;
          pthread_mutex_lock(&mutex);
          switch_active_machine(vm);
          pthread_mutex_unlock(&mutex);
          break;
        }
      }
    }
    //Send the CONTINUE command to VM
    store_last_system_features(&(vm->last_features), current_features);
    vm->last_system_features_stored = 1;
    //sending CONTINUE command to the VM
    bzero(send_buff, BUFSIZE);
    send_buff[0] = CONTINUE;
    if ((send(vm->socket, send_buff, BUFSIZE, 0)) == -1) {
      if (errno == EWOULDBLOCK || errno == EAGAIN) {
        printf("Timeout on send() while sending data by VM %s\n",
            vm->ip);
        fflush(stdout);
      } else {
        printf("Error on send() while sending data by VM %s\n", vm->ip);
        fflush(stdout);
      }
      break;
    }
    sleep(1);
  }
  //If something wrong, do this "exit" operations
  if (close(vm->socket) == 0)
    printf("Connection correctly closed with VM %s\n", vm->ip);
  else
      printf("Error while closing connection with vm %s\n", vm->ip);
  pthread_mutex_lock(&mutex);
  remove_vm_by_ip(vm->ip, &vm_list);
  print_vm_list(vm_list);
  //Notify to LB
  if (vm->state == ACTIVE) {
    strcpy(vm_op.ip, vm->ip);
    vm_op.port = htons(8080);
    vm_op.op = DELETE;
    send_command_to_load_balancer();
  }
  //Active new machine if necessary
  if (get_number_of_active_vms(vm_list) < number_of_active_vm) {
    activate_new_machine();
    print_vm_list(vm_list);
  }
  pthread_mutex_unlock(&mutex);
  pthread_exit(0);
}
```

Listing 5.14: Main thread

Two important functions used in the main thread need to activate new VMs. In particular Listing 5.15 reports the code to rejuvenate a VM then to activate a new VM kept in stand-by.

```c
void switch_active_machine(struct virtual_machine *vm) {
  ...
  //Notify to LB the rejuvenation
  strcpy(vm_op.ip, vm->ip);
  vm_op.port = htons(8080); //TODO
  vm_op.op = DELETE;
  send_command_to_load_balancer();
  //Look for a ready VM
  for (index = 0; index < vm_list_size(vm_list); index++) {
    struct virtual_machine *vm = get_vm_by_position(index, vm_list);
    if (vm->state == STAND_BY) {
      vm->state = ACTIVE;
      printf("Activated vm with ip: %s\n", vm->ip);
      strcpy(vm_op.ip, vm->ip);
      vm_op.port = htons(8080);
      vm_op.op = ADD;
      send_command_to_load_balancer();
      break;
    }
  }
  vm->last_system_features_stored = 0;
  //Send the REJUVENATE command to the VM
  bzero(send_buff, BUFSIZE);
  send_buff[0] = REJUVENATE;
  if ((send(vm->socket, send_buff, BUFSIZE, 0)) == -1) {
    perror("switch_active_machine - send");
    printf("Closing connection with VM %s\n", vm->ip);
    //compact_vm_data_set(vm);
    close(vm->socket);
  } else {
    printf("REJUVENATE command sent to machine with IP address %s\n",
        vm->ip);
    fflush(stdout);
  }
}
```

Listing 5.15: Switching VMs

The second function follows the idea of the just mentioned function with the difference that it is used when a connection with a VM is lost by the controller. It does not need to notify to the load balancer that a VM is crashed because it is performed by the main thread, it has just to activate a new VM and to notify to the load balancer that a new VM is now ready to receive the incoming user' requests. The Listing 5.16 reports the code:

```
void activate_new_machine() {
  int index;
  char send_buff[BUFSIZE];

  for (index = 0; index < vm_list_size(vm_list); index++) {
    struct virtual_machine *vm = get_vm_by_position(index, vm_list);
    if (vm->state == STAND_BY) {
      vm->state = ACTIVE;
      strcpy(vm_op.ip, vm->ip);
      vm_op.port = htons(8080);
      vm_op.op = ADD;
      send_command_to_load_balancer();
      printf("Activated vm with ip: %s\n", vm->ip);
      return;
    }
  }
  printf("No vms available to be activeted\n");
}
```

Listing 5.16: Active new VM

5.4.2 Load Balancer

The Load Balancer is the VM that receives remote user connections to redirect them either to specific VMs in the cloud region or towards remote cloud segments. It represents the entry point of the distributed system from the outside world and it receives from CON informations about which are the currently available VMs. The Load Balancer acts like a slave, following the decisions of the controller and forwarding messages from remote users to a specific VM (or remote cloude region) and vice versa. It lives in the middle between the users and the VMs.

Once the controller is run, it waits for an incoming connection from its own load balancer, then the first step for the load balancer is to connect it to the controller and build up all the needed routines to manage the communication. First of all, two connections are necessary: one to receive infos from the controller related to the management of the VMs in the cloud region and another one to pass to the controller the arrival rate in the cloud region.

```
int main(int argc, char *argv[]) {
  ...
  // CONNECTION LB - CONTROLLER VMs MANAGEMENT
  if (connect(sock_id_controller, (struct sockaddr *) &controller,
      sizeof(controller)) < 0) {
    perror("main: connect_to_controller");
    exit(EXIT_FAILURE);
  }
  // Send to controller balancer public ip address
  get_my_own_ip();
  if (sock_write(sock_id_controller, my_own_ip, 16) < 0) {
    perror(
        "Error in sending balancer public ip address to its own controller:
    ");
  }

  // Once client_sock_id is created, build up a new thread to implement the
    exchange of messages between LB and Controller
  pthread_attr_init(&pthread_custom_attr);
  pthread_create(&tid, &pthread_custom_attr, controller_thread,
      (void *) (long) sock_id_controller);

  // CONNECTION LB - CONTROLLER ARRIVAL RATE
  ...
  if (connect(sock_id_update_region_features, (struct sockaddr *) &
    controller,
      sizeof(controller)) < 0) {
```

```
    perror("main: connect_to_controller arrival rate");
    exit(EXIT_FAILURE);
  }

  memset(regions, 0, sizeof(struct _region) * NUMBER_REGIONS);
  // Once client_sock_id is created, build up a new thread to implement the
    exchange of messages between LB and Controller
  pthread_attr_init(&pthread_custom_attr);
  // Start the timer to evaluate the arrival rate
  timer_start(update_local_region_features_timer);
  pthread_create(&tid_update_region_features, &pthread_custom_attr,
      update_region_features,
      (void *) (long) sock_id_update_region_features);
```

Listing 5.17: Connect to Controller

The first thread manages the communication CON-LB to add and remove VMs, in order to correctely redirect the incoming user requests. In fact, the Load Balancer has to know the active VMs so that it is able to coherently redirect the connections. Listing 5.18 shows how the load balancer acts when a new operation is received from the Controller.

```
void * controller_thread(void * v) {
  ...
  while (1) {
    // Wait for info by the controller
    if ((numbytes = sock_read(socket,&vm_op,sizeof(struct
    virtual_machine_operation))) == -1) {
            ...
        }
    ...
    // Choose what I've to do
    if (vm_op.op == ADD) {
      struct virtual_machine * vm = (struct virtual_machine*) malloc(sizeof(
    struct virtual_machine));
      memcpy(vm->ip, vm_op.ip, 16);
      printf("Adding vm %s\n", vm->ip);
      pthread_mutex_lock(&mutex);
      add_vm(vm, &vm_list);
      printf("------------------\nNew vm list:\n");
```

```
        print_vm_list(vm_list);
        pthread_mutex_unlock(&mutex);
      } else if (vm_op.op == DELETE) {
        printf("Removing vm %s\n", vm_op.ip);
        pthread_mutex_lock(&mutex);
        remove_vm_by_ip(vm_op.ip, &vm_list);
        printf("------------------\nNew vm list:\n");
        print_vm_list(vm_list);
        pthread_mutex_unlock(&mutex);
      } else if (vm_op.op == REJ) {
        printf("Removing vm %s\n", vm_op.ip);
        pthread_mutex_lock(&mutex);
        remove_vm_by_ip(vm_op.ip, &vm_list);
        printf("------------------\nNew vm list:\n");
        print_vm_list(vm_list);
        pthread_mutex_unlock(&mutex);
      } else {
        // something wrong, operation not supported!
        printf("Received not supported operation from controller\n");
      }
    }
}
```

Listing 5.18: Thread to communicate with the Controller

The second thread manages the connection CON-LB to send the arrival rate computed by the balancer. The arrival rate λ is computed considering a fixed time interval, defined as a macro, as shown in Listing 5.19. The load balancer has to wait for the forwarding probability sent from the leader, according to the code already seen in Listing 5.11. Finally the timer is restarted.

```
void *update_region_features(void * sock) {
  ...
  while (1) {

    // Compute the arrival rate, then send it
    double time = timer_value_seconds(update_local_region_features_timer);
    float local_region_user_request_arrival_rate = (float) lambda
```

```
            / (float) time;
        if (sock_write(sockfd, &local_region_user_request_arrival_rate,
            sizeof(float)) < 0)
          perror("Error in writing local arrival rate to controller");
        memset(temp_regions, 0, sizeof(struct _region) * NUMBER_REGIONS);

        // Wait for the probability from the leader
        if (sock_read(sockfd, &temp_regions,
            sizeof(struct _region) * NUMBER_REGIONS) < 0) {
          perror("Error in reading probabilities from the leader");
        }
        //pthread_mutex_lock(&mutex);
        memcpy(&regions, &temp_regions,
            sizeof(struct _region) * NUMBER_REGIONS);

        // Restart the timer and reset the arrival rate value
        timer_restart(update_local_region_features_timer);
        lambda = 0;

        // Wake me up after UPDATE_LOCAL_REGION_FEATURE_INTERVAL
        while (timer_value_seconds(update_local_region_features_timer)
            < UPDATE_LOCAL_REGION_FEATURE_INTERVAL) {
          sleep(1);
        }
      }
    }
```

Listing 5.19: Sending the arrival rate value

The next step allows to the load balancer to process incoming requests from remote cloud regions. As already seen, the main goal of this distributed application is to allow to transparently redirect user incoming connections towards any region, depending on the current load and state of accumulation of anomalies in all the cloud regions by relying on MTTF value. So, the load balancer has to accept for remote load balancer requests, and it has to manage these incoming connections as they are its local connections.

```c
void * accept_balancers(void * v) {
  while (1) {
    ...
    client_sock_id = accept(socket_remote_balancer,
        (struct sockaddr *) &client, &addr_len);
    ...
    // Set non blocking the socket
    setnonblocking(client_sock_id);

    struct arg_thread *vm_client = (struct arg_thread*) malloc(
        sizeof(struct arg_thread));

    // Manage the new incoming requests
    vm_client->socket = client_sock_id;
    strcpy(vm_client->ip_address, inet_ntoa(client.sin_addr));
    vm_client->port = ntohs(client.sin_port);
    // Mark this connection as received from a remote LB
    vm_client->user_type = 1;

    // Manage the connection USER-VM
    res_thread = create_thread(client_sock_id_thread, vm_client);
  }
}
```

Listing 5.20: Accepting load balancer remote requests

In the just showed Listing 5.20, it is introduced a thread function to manage the communication between users and VMs. This thread is invoked by both the leader and by the main thread using two different values for representing the kind of connection. In particular, if a connection is received from a remote load balancer, the load balancer has to exclude the user request in computing the arrival rate. On the other hand, when an incoming connection is directely received by the load balancer, the user request has to be included in the computation. Listing 5.21 reports the differences just mentioned in invoking the thread to manage the connection among users and Vms.

```c
...
struct arg_thread *vm_client = (struct arg_thread*) malloc(sizeof(struct
    arg_thread));
vm_client->socket = client_sock_id;
strcpy(vm_client->ip_address, inet_ntoa(client.sin_addr));
vm_client->port = ntohs(client.sin_port);
vm_client->user_type = 0;

res_thread = create_thread(client_sock_id_thread, vm_client);
...
```

Listing 5.21: Manage incoming user requests

Finally the code implements the last step for the load balancer: allow the communication between users and VMs that are providing a service. Initially, it has to be got the sockets for the user and for a VM. They are used to create a descriptor set monitored by a select function. Listing 5.22 shows the select function signature.

```c
int select(int nfds, fd_set *restrict readfds, fd_set *restrict writefds,
    fd_set *restrict errorfds, struct timeval *restrict timeout);
```

Listing 5.22: Select function signature

The implementation of the load balancer needs non-blocking sockets that have to be monitored simultaneasly. The Select() system call examines the I/O descriptor sets whose addresses are passed in readfds, writefds, and errorfds to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first nfds descriptors are checked in each set; i.e., the descriptors from 0 through nfds-1 in the descriptor sets are examined. The framework puts both the sockets (e.g. client socket and vm socket) in a descriptor set monitored both for reading and for writing operation. Select() returns the number of ready descriptors that are contained in the descriptor sets, or -1 if an error occurred. If the time limit expires, select() returns 0. In this context, a socket may be ready for a reading operation and for a writing operation at the same time. To avoid a deadlock, when the select returns, it is checked if the set contains the client socket, then it is checked if there are some bytes ready for the client and they are written to the client. To avoid the above mentioned deadlock, a read operation is always tried and the incoming bytes are appended to an aux buffer until another write operation will be permormed. The same for the vm socket. To better understand how the code works, see the Listing 5.23.

```c
void *client_sock_id_thread(void *vm_client_arg) {
  ...
  // Assign the sockets
  int client_socket = vm_client.socket;
  int vm_socket = create_socket(vm_client.ip_address, vm_client.port,
    vm_client.user_type);
  ...
  // We never finish forwarding data!
  while (1) {
    ...
    // Setup a timeout
    timeout.tv_sec = 1;
    timeout.tv_usec = 0;

    readsocks = select(highsock + 1, &socks, &socks, (fd_set *) 0,
        &timeout);
    ...
    } else {

      // Check if the client is ready
      if (FD_ISSET(client_socket, &socks)) {
        // First of all, check if we have something for the client
        if (bytes_ready_to_client > 0) {
          if ((transferred_bytes = sock_write(client_socket,
              buffer_to_client, bytes_ready_to_client)) < 0) {
            perror("write: sending to client from buffer");
          }

          bytes_ready_to_client = 0;
          bzero(buffer_to_client, FORWARD_BUFFER_SIZE);
        }

        // Always perform sock_read, if it returns a number greater than
    zero
        // something has been read then we've to append aux buffer to
    previous buffer (pointers)
        transferred_bytes = sock_read(client_socket,
            aux_buffer_from_client, FORWARD_BUFFER_SIZE);
        ...

        if (transferred_bytes > 0) {
          append_buffer(buffer_from_client, aux_buffer_from_client,
              &bytes_ready_from_client, transferred_bytes,
              &times);
        }

        // Increment the number of incoming requests! Used to compute the
    arrival rate
        if (bytes_ready_from_client > 0) {
          if (!vm_client.user_type)
            lambda++;
        }

      }
      // Check if the VM is ready
      if (FD_ISSET(vm_socket, &socks)) {
        // First of all, check if we have something for the VM to send
        if (bytes_ready_from_client > 0) {
          if ((transferred_bytes = sock_write(vm_socket,
              buffer_from_client, bytes_ready_from_client)) < 0) {
            perror("write: sending to vm from client");
          }
          bytes_ready_from_client = 0;
          bzero(buffer_from_client, FORWARD_BUFFER_SIZE);
        }

        // Always perform sock_read, if it returns a number greater than
    zero
        // something has been read then we've to append aux buffer to
    previous buffer (pointers)
        transferred_bytes = sock_read(vm_socket, aux_buffer_to_client,
            FORWARD_BUFFER_SIZE);
        ...

        if (transferred_bytes > 0) {
          append_buffer(buffer_to_client, aux_buffer_to_client,
              &bytes_ready_to_client, transferred_bytes, &times);
        }
      }
    }
  }
}
```

Listing 5.23: Thread to manage the connections

For the sake of clearness, the following code shows how the function to append to a buffer works.

```
void append_buffer(char * original_buffer, char * aux_buffer,
    int * bytes_original, int bytes_aux, int * times) {

  if ((*bytes_original + bytes_aux) >= FORWARD_BUFFER_SIZE) {
    printf("REALLOC: STAMPA TIMES: %d\n", *times);
    original_buffer = realloc(original_buffer,
        (*times) * FORWARD_BUFFER_SIZE);
    (*times)++;
  }
  memcpy(&(original_buffer[*bytes_original]), aux_buffer, bytes_aux);
  *bytes_original = *bytes_original + bytes_aux;
  bzero(aux_buffer, FORWARD_BUFFER_SIZE);
}
```

Listing 5.24: Append buffer function

Due to the fact that the Load Balancer has to rely on the probabilities received from the leader, it follows that ad-hoc manner is implemented to forward the incoming user connections either towards the VMs in the region or towards a remote load balancer. In the following two Listings 5.26 and 5.27

it is represented the forwarding policy relying on the probabilities.

```
int create_socket(char * ip_client, int port_client, int user_type) {
  int sock_id = socket(AF_INET, SOCK_STREAM, 0);
  ...
  struct sockaddr_in saddr = get_target_server_saddr(ip_client, port_client,
      user_type);
  if (connect(sock_id, (struct sockaddr *) &saddr, sizeof(saddr)) < 0) {
    perror("Error while connecting socket for new client");
    return 0;
  }
  setnonblocking(sock_id);
  return sock_id;
}
```

Listing 5.25: Aux function to create the client side connection

```
struct sockaddr_in get_target_server_saddr(char * ip, int port, int
    user_type) {
  ...
  switch (user_type) {

  case 0: //from a user
    probability_sum = 0;
    random = (float) rand() / (float) RAND_MAX;
    index = 0;
    probability_sum = regions[index].probability;
    while (random > probability_sum && index < NUMBER_REGIONS) {
      index++;
      probability_sum += regions[index].probability;
    }
    if (!strcmp(regions[index].ip_balancer, my_own_ip) || index ==
    NUMBER_REGIONS) {
      target_server_saddr = select_local_vm_addr();
      return target_server_saddr;
    } else {
      target_server_saddr.sin_addr.s_addr = inet_addr(regions[index].
    ip_balancer);
      target_server_saddr.sin_port = htons(port_remote_balancer);
      return target_server_saddr;
    }

  case 1: //from a remote balancer
    target_server_saddr = select_local_vm_addr();
    return target_server_saddr;
  }
}
```

Listing 5.26: Choose forwarding strategy

```
struct sockaddr_in select_local_vm_addr() {
  static int current_rr_index = 0;
  struct sockaddr_in target_vm_saddr;
  target_vm_saddr.sin_family = AF_INET;
  if (vm_list_size(vm_list) == 0)
    return target_vm_saddr;
  if (current_rr_index >= vm_list_size(vm_list))
    current_rr_index = 0;
  struct virtual_machine *vm = get_vm_by_position(current_rr_index, vm_list)
    ;
  current_rr_index++;
  target_vm_saddr.sin_addr.s_addr = inet_addr(vm->ip);
  target_vm_saddr.sin_port = htons(VM_SERVICE_PORT);
  return target_vm_saddr;
}
```

Listing 5.27: Local forwarding

5.4.3 VM

The VM represents a machine that is hosting a copy of the application, in this context it is the TPC-W transactional web benchmark[8]. In this framework the application runs beside a process for getting the system features. In particular a connection between the controller and the VM has to be established, in which the mentioned process has to communicates the read measurements. It does not know anything about the read system values, but it has to wait from the controller the command to either continue or rejuvenate the current session. Due to this fact, the only interesting function in the VM is to get the features, as shown in the Listing 5.28 and 5.29.

```c
void get_features(char * output){
  ...
  // Monitored system features
    int mem_total, mem_used, mem_free, mem_shared, mem_buffers, mem_cached;
    int swap_total, swap_used, swap_free;
    unsigned long cpu_user1, cpu_nice1, cpu_system1, cpu_iowait1, cpu_steal1
    , cpu_idle1, dummyA1, dummyB1, dummyC1, dummyD1;
    unsigned long cpu_user2, cpu_nice2, cpu_system2, cpu_iowait2, cpu_steal2
    , cpu_idle2, dummyA2, dummyB2, dummyC2, dummyD2;
    float cpu_user, cpu_nice, cpu_system, cpu_iowait, cpu_steal, cpu_idle;
    float cpu_total;
    struct timeval curr_time;

    char num_th[128];
    FILE *pof, *fstat, *fmem;

    // Get number of active threads
    pof = popen("ps -eLf | grep -v defunct | wc -l", "r");
    if(pof == NULL)
        abort();

    fgets(num_th, sizeof(num_th)-1, pof);
    pclose(pof);

    // Get timestamp
    gettimeofday(&curr_time, NULL);
    sprintf(output, "Datapoint: %f %s", (double)curr_time.tv_sec -
    initial_time.tv_sec + (double)curr_time.tv_usec / 1000000 - (double)
    initial_time.tv_usec / 1000000, num_th);

    // Get memory and swap usage (using /proc/meminfo)
    t = fopen("/proc/meminfo", "r");
    if (t == NULL) {
        perror("FOPEN ERROR MEMINFO ");
        exit(EXIT_FAILURE);
    }
    index = 0;
    bzero(aux_buffer,BUFSIZE);
    while ((ch = fgetc(t)) != EOF) {
        aux_buffer[index++] = ch;
    }
    fclose(t);
    sscanf(aux_buffer,"MemTotal: %d kB MemFree: %d kB Buffers: %d kB Cached:
 %d kB", &mem_total, &mem_free, &mem_buffers, &mem_cached);
    mem_used = mem_total − mem_free;
    mem_shared = 0;
    sprintf(output, "%sMemory: %d %d %d %d %d %d\n", output, mem_total,
    mem_used, mem_free, mem_shared, mem_buffers, mem_cached);
    pointer_buffer = strstr(aux_buffer,"SwapTotal:");
    sscanf(pointer_buffer,"SwapTotal: %d kB SwapFree: %d kB", &swap_total, &
    swap_free);
    swap_used = swap_total − swap_free;
    sprintf(output, "%sSwap: %d %d %d\n", output, swap_total, swap_used,
    swap_free);
```

Listing 5.28: Get the system features - Memory

```
    // Get CPU Usage (using /proc/stat)
    fstat = fopen("/proc/stat", "r");
    if (fstat == NULL) {
        perror("FOPEN ERROR FIRST /PROC/STAT ");
        exit(EXIT_FAILURE);
    }
    if (fscanf(fstat, "%s %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu", \&cpu\
    _user1, \&cpu\_nice1,
    \&cpu\_system1, \&cpu\_idle1, \&cpu\_iowait1, \&dummyA1, \&dummyB1, \&
    cpu\_steal1, \&dummyC1, \&dummyD1) == EOF) {
        exit(EXIT\_FAILURE);
    }
    fclose(fstat);
    sleep(1);
    fstat = fopen("/proc/stat", "r");
    if (fstat == NULL) {
        perror("FOPEN ERROR SECOND /PROC/STAT ");
        exit(EXIT\_FAILURE);
    }
    if (fscanf(fstat, "%s %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu", \&cpu\
    _user2, \&cpu\_nice2,
    \&cpu\_system2, \&cpu\_idle2, \&cpu\_iowait2, \&dummyA2, \&dummyB2, \&
    cpu\_steal2, \&dummyC2, \&dummyD2) == EOF) {
        exit(EXIT\_FAILURE);
    }
    fclose(fstat);
    cpu\_total = (float)(cpu\_user2 + cpu\_nice2 + cpu\_system2 + cpu\_idle2
     + cpu\_iowait2 + dummyA2 + dummyB2 + cpu\_steal2 + dummyC2 + dummyD2);
    cpu\_total -= (float)(cpu\_user1 + cpu\_nice1 + cpu\_system1 + cpu\
    _idle1 + cpu\_iowait1 + dummyA1 + dummyB1 + cpu\_steal1 + dummyC1 +
    dummyD1);

    cpu\_user = (float)(cpu\_user2 - cpu\_user1) * 100.0 / cpu\_total;
    cpu\_system = (float)(cpu\_system2 - cpu\_system1) * 100.0 / cpu\_total;
    cpu\_nice = (float)(cpu\_nice2 - cpu\_nice1) * 100.0 / cpu\_total;
    cpu\_idle = (float)(cpu\_idle2 - cpu\_idle1) * 100.0 / cpu\_total;
    cpu\_iowait = (float)(cpu\_iowait2 - cpu\_iowait1) * 100.0 / cpu\_total;
    cpu\_steal = (float)(cpu\_steal2 - cpu\_steal1) * 100.0 / cpu\_total;

    sprintf(output,"%sCPU: %f %f %f %f %f %f", output, cpu\_user, cpu\_nice,
     cpu\_system, cpu\_iowait, cpu\_steal, cpu\_idle);
    return;
}
```

Listing 5.29: Get the system features - CPU

Based on this features, the controller may be decide to rejuvenate the VM, in this case the machine is rebooted after a certain time to allow to the machine to serve the pending requests.

```
...
if (recv_buff[0]==REJUVENATE) {
  //wait for completing pending requests
  printf("Command REJUVENATE received\n");
  printf("Waiting %d seconds for completing pending requests before
    rejuvenation...\n", TIME_FOR_COMPLETING_PENDING_REQUESTS);
  fflush(stdout);
  sleep(TIME_FOR_COMPLETING_PENDING_REQUESTS);
  printf("Executing reboot...\n");
  fflush(stdout);
  system("reboot");
  exit(0);
}
...
```

Listing 5.30: Rejuvenate the VM

5.4.4 User

In this framework is supposed that the user is a common user surfing on the web. To simulate incoming user requests, it is chosen to use emulated browsers provided by TPC-W. A more detailed description will be provided in Chapter 6, where it is showed the experimental evaluation and all the choices related to the achieved results.

Chapter 6

Experimental evaluation

The experiments is carried out using three different regions:

1. Region 1 is hosted by Amazon EC2 services in Ireland. The used instances in this region are *m3.medium*. This kind of instances use Intel Xeon E5-2670 v2 (Ivy Bridge) processors and SSD storage. In particular they have 1 vCPU, 3.75GiB of Memory and a storage SSD of 1 x 4 GB.

2. Region 2 is hosted by Amazon EC2 services in Frankfurt. The used instances in this region are *m3.medium*.

3. Region 3 is privately hosted in HP ProLiant server in Munich. The used hypervisor if Vmware Workstation 10.4.

All virtual machines of the experimental environment were equipped with Ubuntu 10.04 Linux Distribution (kernel version 2.6.32-5-amd64). This set up is representative of a hybrid virtualized environment.

The test-bed application is a multi-tier e-commerce web application that simulates an on-line book store, following the standard configuration of TPC-

W benchmark[8] and using the Java implementation[25], developed using servlets, and relying on MySql[26] as a date base server.

In order to generate TPC-W requests, we have used an emulated browser. To evaluate the RT of the web application, software probes are introduced in the emulated browsers to store on a database file the response time of every web interaction.

The TPC-W implementation has been modified in order to generate the anomalies of interest for this experimentation, namely memory leaks and unterminated threads. Specifically, the *Home Web Interaction* class has been modified (it implements the beginning of a TPC-W session) so that, when the servlet is started up, two different rates (for memory leaks and unterminated threads) are generated. Then, whenever an emulated browser connects to the initial page, some memory is leaked or a new thread is spawn, according to the corresponding probability. By using this approach, the generation rate of anomalies directly depends on the load of the TPC-W server (i.e on the number of connections by the emulated browsers). TPC-W clients are connected to load balancers of regions 1,2 and 3, and the number of active clients (towards each region) varies in the interval [16,512].

Tables 6.1 and 6.2 report respectively, the Soft-Mean Absolute preditction Error (S-MAE) for Amazon and Munich regions.

The experiment was performed in a time interval of about 60 when two regions are considered, and about 120 minutes when three regions are considered, with atmost six active VMs per region keeping the other in $STAND\_BY$ state. Due to the framework design, it is not important which of the two regions on three are activated, this framework is able to work on hybrid cloud environment.

Table 6.1: Soft Mean Absolute Error—10% Threshold (Amazon regions)

Using all parameters		**Using only parameters selected by Lasso**	
Algorithm	**Error (seconds)**	**Algorithm**	**Error (seconds)**
Linear Regression	137.600	Linear Regression	156.603
M5P	79.182	M5P	118.292
REP Tree	69.832	REP Tree	108.476
SVM	132.668	SVM	146.594
SVM2	132.675	SVM2	146.607
Lasso ($\lambda = 10^0$)	405.187	Lasso ($\lambda = 10^0$)	405.187
Lasso ($\lambda = 10^1$)	405.187	Lasso ($\lambda = 10^1$)	405.187
Lasso ($\lambda = 10^2$)	405.186	Lasso ($\lambda = 10^2$)	405.186
Lasso ($\lambda = 10^3$)	405.178	Lasso ($\lambda = 10^3$)	405.178
Lasso ($\lambda = 10^4$)	405.124	Lasso ($\lambda = 10^4$)	405.124
Lasso ($\lambda = 10^5$)	404.823	Lasso ($\lambda = 10^5$)	404.823
Lasso ($\lambda = 10^6$)	404.041	Lasso ($\lambda = 10^6$)	404.041
Lasso ($\lambda = 10^7$)	399.023	Lasso ($\lambda = 10^7$)	399.023
Lasso ($\lambda = 10^8$)	399.240	Lasso ($\lambda = 10^8$)	399.240
Lasso ($\lambda = 10^9$)	392.469	Lasso ($\lambda = 10^9$)	392.469

Table 6.2: Soft Mean Absolute Error—10% Threshold (Munich region)

Using all parameters		**Using only parameters selected by Lasso**	
Algorithm	**Error (seconds)**	**Algorithm**	**Error (seconds)**
Linear Regression	125.421	Linear Regression	132.127
M5P	76.623	M5P	116.393
REP Tree	68.923	REP Tree	112.932
SVM	139.012	SVM	134.282
SVM2	131.053	SVM2	133.220
Lasso ($\lambda = 10^0$)	399.109	Lasso ($\lambda = 10^0$)	399.109
Lasso ($\lambda = 10^1$)	399.107	Lasso ($\lambda = 10^1$)	399.107
Lasso ($\lambda = 10^2$)	399.115	Lasso ($\lambda = 10^2$)	399.115
Lasso ($\lambda = 10^3$)	399.103	Lasso ($\lambda = 10^3$)	399.103
Lasso ($\lambda = 10^4$)	399.101	Lasso ($\lambda = 10^4$)	399.101
Lasso ($\lambda = 10^5$)	399.021	Lasso ($\lambda = 10^5$)	399.021
Lasso ($\lambda = 10^6$)	399.001	Lasso ($\lambda = 10^6$)	399.001
Lasso ($\lambda = 10^7$)	397.923	Lasso ($\lambda = 10^7$)	397.923
Lasso ($\lambda = 10^8$)	397.872	Lasso ($\lambda = 10^8$)	397.872
Lasso ($\lambda = 10^9$)	391.126	Lasso ($\lambda = 10^9$)	391.126

The Figure 6.1, shows how the MTTF and the Forwarding Probability values change when one then two regions are activated and a constant request rate of 500 req/s come. In particular, initially, all the traffic is served by the unique active region. When a new region is added to the system (i.e. 22 minutes in the execution time), the incoming requests are forwarded towards both the two regions in order to increase the global MTTF value, reaching a steady-state situation where the MTTF values of the two regions are the similar. Of course this result is tied to the forwarding probabilities. The plot at the top of the Figure 6.1, shows how the MTTF values change when a new region is added and the framework divides the incoming traffic. Of course, when a region is removed (i.e. 54 minutes in the execution time), only one region is active and is able to serve the incoming requests, then its MTTF value has to decrease.

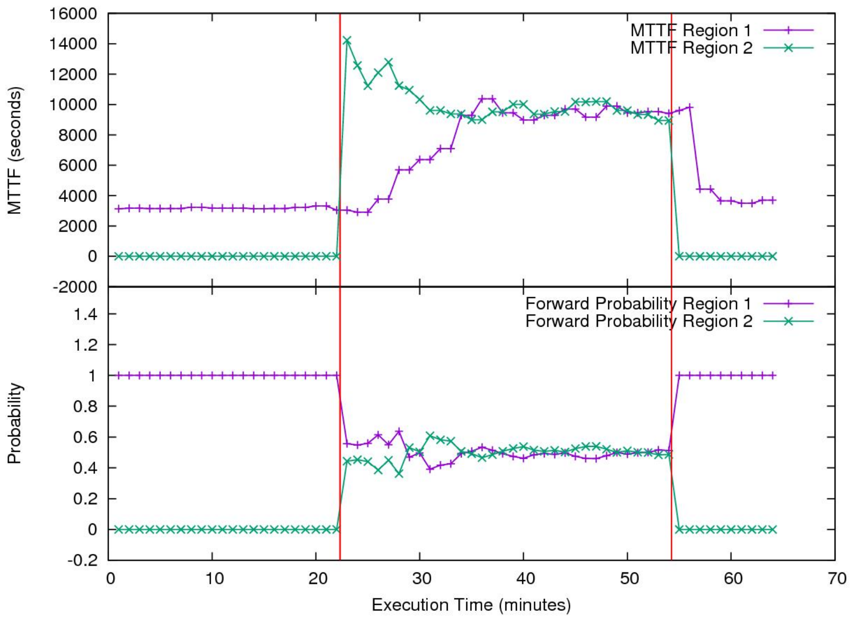


Fig. 6.1: Results using two regions

The main results are represented in the Figure 6.2. This experiment run for 130 minutes, varying the requests rate and using at least two regions, then activating a new region, trying to simulate a real scenario. The plot at the bottom, simply represents how the requests rate varies over the time. The two above plots, represent how the forwarding probabilities and MTTF values change when the number of the requests per second change. In particular, until 22 minutes, the rate is constant at 300 req/sec and the forwarding probabilities of the two active regions and them own MTTF values are similar. When the incoming rate increase up to 700 req/s, but again only two regions are active, the forwarding probabilities remain the same, but the MTTF values decrease, due to the accumulation of anomalies, namely increase in memory leaks and unterminated threads rates. Finally, to evaluate properly this framework, a new region is added to the system. First of all, the system balances the forwarding probabilities in order to exploit the just added region. Due to this fact, the workload is distributed over one more region, allowing to the system to increase the global MTTF value and proving that the proposed load balancing strategy in Section 5.3 works in hybrid cloud environments.

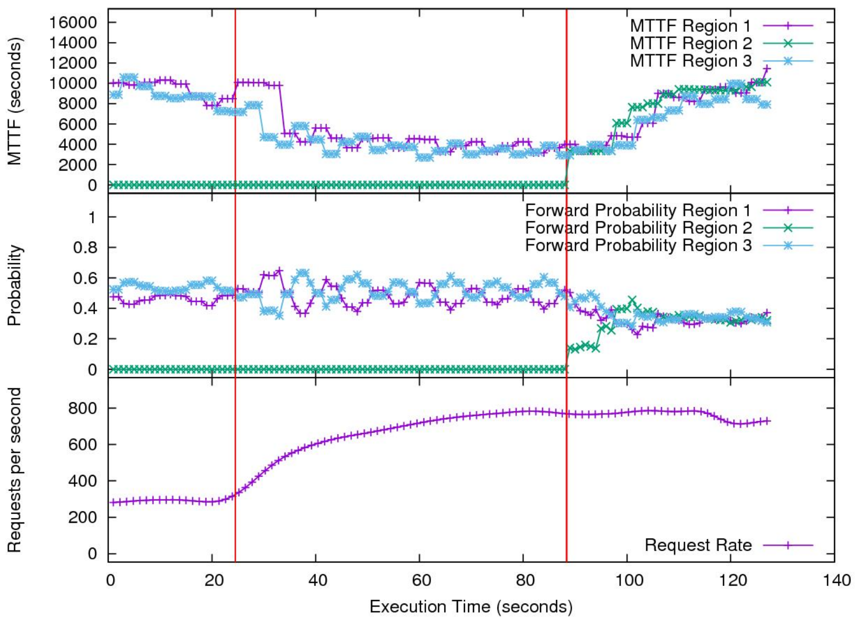


Fig. 6.2: Results using three regions

Another interesting result is the alternance in the MTTF values and forwarding probabilities. Indeed, the framework always tries to balance these values to reach a steady state situation, so when the region 1 MTTF is greater than the region 2 MTTF, the framework tries to send more incoming traffic fraction towards the region 1, causing a fluctuation in the forwarding probability then in the MTTF values, and leading the region 2 MTTF value to be greater than the region 1 MTTF, and so on. After a certain time, the system becomes stable, as shown in the just seen figures.

Chapter 7

Conclusions

In this work, an innovative framework to perform a proactive workload management in multi-cloud environment is presented. This framework exploits the PCAM Machine Learning-based framework to predict the *Remaining Time To Failure* (RTTF) value for the VMs composing a cloud region, in order to face the problems related to the software aging, and ensuring an increase in the cloud segment availability applying a proactive rejuvenation procedure. Again, an innovative load balancing strategy is presented relying on the *Mean Time To Failure* (MTTF) value for each region. Due to the fact that the framework uses OS's parameter measurements (e.g. free memory, cpu usage, etc...) and network monitoring (e.g. response time, arrival rate), it is able to work independently from the underlying machines. The combination of the rejuvenation strategy and the load balancing strategy allows the increase of the system's availability, then the decrease of the response time (user side) and the possibility to simulate the behavior of the most common distributed applications. The experiment evaluation shows that when two regions are active, triggering the procedure to add a new third region improves the MTTF value 2.5 times (i.e. from 4000 seconds to 10000 seconds) leading the whole the distributed system to guarantee an higher availability.

This work has not considered the overheads and delays due to the network. They may be analyzed in future works, in order to better represent real scenarios and nowadays common distributed applications.

Chapter 8

Bibliography

[1] D. L. Parnas, "Software aging," in *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, (Los Alamitos, CA, USA), pp. 279–287, IEEE Computer Society Press, 1994.

[2] S. Pertet and P. Narasimhan, "Causes of failure in web applications (cmu-pdl-05-109)," *Parallel Data Laboratory*, p. 48, 2005.

[3] K. S. Trivedi, M. Grottke, and E. Andrade, "Software fault mitigation and availability assurance techniques," *International Journal of System Assurance Engineering and Management*, vol. 1, no. 4, pp. 340–350, 2010.

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

[5] P. Di Sanzo, A. Pellegrini, and D. R. Avresky, "Machine learning for achieving self-* properties and seamless execution of applications in

the cloud," in *Proceedings of the Fourth IEEE Symposium on Network Cloud Computing and Applications*, NCCA, IEEE Computer Society, June 2015.

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

[7] D. Avresky and N. Natchev, "Dynamic reconfiguration in computer clusters with irregular topologies in the presence of multiple node and link failures," *Computers, IEEE Transactions on*, vol. 54, no. 5, pp. 603–615, 2005.

[8] W. D. Smith, "Tpc-w: Benchmarking an ecommerce solution," 2000.

[9] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Software aging and rejuvenation: Where we are and where we are going," in *Proceedings - 2011 3rd International Workshop on Software Aging and Rejuvenation, WoSAR 2011*, pp. 1–6, 2011.

[10] H. Zhang, G. Jiang, K. Yoshihira, and H. Chen, "Proactive Workload Management in Hybrid Cloud Computing," 2014.

[11] Y. Han and A. T. Chronopoulos, "A Resilient Hierarchical Distributed Loop Self-Scheduling Scheme for Cloud Systems," in *IEEE 13th International Symposium on Network Computing and Applications*, 2014.

[12] D. Ardagna, S. Casolari, M. Colajanni, and B. Panicucci, "Dual time-scale distributed capacity allocation and load redirect algorithms for

cloud systems," *Journal of Parallel and Distributed Computing*, vol. 72, no. 6, pp. 796–808, 2012.

[13] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software - Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.

[14] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Proceedings - 2011 IEEE 4th International Conference on Cloud Computing, CLOUD 2011*, pp. 500–507, 2011.

[15] R. N. Calheiros, R. Ranjan, and R. Buyya, "Virtual Machine Provisioning Based on Analytical Performance and QoS in Cloud Computing Environments," *2011 International Conference on Parallel Processing*, pp. 295–304, 2011.

[16] S. Pacheco-Sanchez, G. Casale, B. Scotney, S. McClean, G. Parr, and S. Dawson, "Markovian workload characterization for QoS prediction in the cloud," in *Proceedings - 2011 IEEE 4th International Conference on Cloud Computing, CLOUD 2011*, pp. 147–154, 2011.

[17] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson, "Statistics-driven workload modeling for the cloud," in *Proceedings - International Conference on Data Engineering*, pp. 87–92, 2010.

[18] P. Di Sanzo, F. Quaglia, B. Ciciani, A. Pellegrini, D. Didona, P. Romano, R. Palmieri, and S. Peluso, "A Flexible Framework for Accurate Simu-

lation of Cloud In-Memory Data Stores," *Simulation Modelling Practice and Theory*, 2015.

[19] M. Grottke, R. Matias, and K. S. Trivedi, "The fundamentals of software aging," in *Software Reliability Engineering Workshops, 2008. ISSRE Wksp 2008. IEEE International Conference on*, pp. 1–6, IEEE, 2008.

[20] M. Grottke and K. S. Trivedi, "Fighting bugs: Remove, retry, replicate, and rejuvenate," *Computer*, vol. 40, no. 2, pp. 107–109, 2007.

[21] D. Simeonov and D. Avresky, "Proactive software rejuvenation based on machine learning techniques," in *Cloud Computing*, pp. 186–200, Springer, 2010.

[22] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288, 1996.

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

[24] E. Alpaydin, *Introduction to machine learning.* MIT press, 2014.

[25] T. Bezenek, T. Cain, R. Dickson, T. Heil, M. Martin, C. McCurdy, R. Rajwar, E. Weglarz, C. Zilles, and M. Lipasti, "Characterizing a java implementation of tpc-w," in *Third Workshop On Computer Architecture Evaluation Using Commercial Workloads*, 2000.

[26] A. MySQL, "Mysql database server," *Internet WWW page, at URL: http://www. mysql. com (last accessed/1/00)*, 2004.

Chapter 9

Ringraziamenti

Con questo progetto di tesi, si chiude un capitolo della mia vita che ricorderò con grandissimo piacere. Il percorso non è stato dei più facili, e come tutte le cose, le difficoltà non si sono fatte attendere. Durante questi anni ho avuto la fortuna di essere circondato da persone veramente eccezionali, per cui, una sola paginetta di ringraziamenti non può bastare. Non me ne vorranno coloro che non nominerò direttamente, ma chi c'è stato, sicuramente lo sa, e si riconoscerà in questi più che dovuti ringraziamenti.

Il mio ringraziamento più grande va ai miei genitori, che al di là del sostegno economico, sono stati sempre presenti, soprattutto nei momenti di difficoltà, insegnandomi che quando si pensa di aver toccato il fondo ci si deve rialzare, e lottare, con la tenacia di voler raggiungere l'obiettivo prefissato. Un ringraziamento speciale va a mio fratello, perchè il suo spirito critico è sempre stato per me fonte di ispirazione e di emulazione nel cercare di superare la superficialità delle cose. Non posso non citare Assunta e il mio nipotino Simone, perchè hanno portato una ventata di felicità alla nostra famiglia, contribuendo sempre a rendere più leggere le giornate e per essere una piacevolissima fuga dalla quotidianeità.

A Edda faccio un grandissimo ringraziamento per essere sempre un punto saldo, con la sua disponibilità e soprattutto pazienza quando mi approcciavo verso la fine di questo percorso. Inoltre non posso non mandarle un enorme abbraccio, per esprimergli tutta la mia stima per il suo coraggio, mentre agli altri sono capitato, lei me s'è proprio scelto!

Agli amici di sempre Francesco, Aurelio e Giuseppe faccio i più sentiti ringraziamenti e scuse nel contempo, per esserci sempre stati e aver sempre dato quel pizzico di colore nonostante le continue buche (anche dell'ultimo momento), rendendo questi anni veramente indimenticabili, regalandomi emozioni indelebili dentro di me.

Non me ne vorranno i "Fracichi di Ingegneria" se un ringraziamento a parte lo faccio a Mauro, compagno di numerosi progetti ed esami preparati insieme, perchè è la persona più brillante che abbia mai conosciuto e che oltre alla sua amicizia, mi ha donato sempre prospettive mai banali per analizzare le situazioni che abbiamo dovuto affrontare, in ambito accademico e non. A tutti gli altri fracichi, Guerino, Francesco, Elisa, Federica, Marcella, Celeste (anche detti "Gli amici di Ninodimmerda") un grazie che viene dal cuore. Senza di voi questo percorso non avrebbe avuto lo stesso epilogo, sicuramente non nella spensieratezza con il quale mi avete portato ad affrontare situazioni anche complicate. Siete sempre stati grandissimi amici e ottimi compagni di studio, e non avrei mai sperato di avere tanta fortuna.

Agli amici del fantacalcio, Giorgio, Corrado, Francesco, Alessandro e Simone va un ringraziamento per avermi spilato un sacco di soldi, concedendomi di essere il Re di Coppa e perché no, per avermi regalato un pò di spensieratezza, che non fa mai male.

Ultimi non per importanza, ringrazio di cuore il mio relatore, il Prof. Bruno Ciciani, perchè oltre che ad insegnarmi nozioni tecniche, ha sempre tentato di trasmettermi valori di cui farò tesoro nel mio futuro, e perchè mi ha concesso la possibilità di poter collaborare in questi mesi con un gruppo fantastico, che forse, mi ha insegnato più cose di quanto un intero corso di laurea ha potuto insegnarmi. Per questo motivo i ringraziamenti non saranno mai abbastanza per Pierangelo Di Sanzo e Alessandro Pellegrini, perchè sempre pazientemente mi hanno guidato, rispondendo ad ogni mio dubbio, e facendomi sempre sentire parte integrante di un team, anche quando ci conoscevamo appena.

Alla fine di questi ringraziamenti, una dedica speciale va a mia madre, che più di ogni altro mi ha insegnato lo spirito di sacrificio, e quanto le cose semplici della vita, a volte sono le più belle. A mio padre una dedica soprattutto per il suo "Cerca a strigne!", che ricorderò fin quando avrò memoria, e per avermi insegnato la serietà e la dedizione in quello che faccio e che farò.

Chiudo ringraziando Igor, perchè è d'obbligo dopo che Giorgio non l'ha fatto nella sua tesi.