A Fuzzy Load Balancer for Adaptive Fault Tolerance Management in Cloud Platforms

Hamid Arabnejad[1], Claus Pahl[2], Giovani Estrada[3], Areeg Samir[2], and Frank Fowley[1]

[1] IC4, Dublin City University, Dublin, Ireland,
[2] Free University of Bozen-Bolzano, Bolzano, Italy,
[3] Intel, Leixlip, Ireland

Abstract. To achieve high levels of reliability, availability and performance in cloud environments, a fault tolerance approach to handle failures effectively is needed. In most existing research, the primary focus has been on explicit specification-driven solutions which requires too much effort for application developers, and leads to inflexibility. We propose a fuzzy job distributor (load balancer) for fault tolerance management to reduce levels of management complexity for the user. The proposed approach aims to *reduce* the possibility of *fault occurrences* in the system by a fair distribution of user job requests among available resources. In our self-adaptive approach, the system manages anomalous situations that might lead to failure by distributing the incoming job request based on the reliability of processing nodes, i.e., virtual machines (VMs). The reliability of VMs is a variable parameter and changes during its lifetime. Our approach is implemented and comparatively analysed using OpenStack. The experimental results show a significant reduction in the occurrence of faults in comparison with other load balancing algorithms.

Keywords: Load Balancing, Job distributor, Fault tolerance, Fuzzy logic, Cloud computing, Anomaly detection, OpenStack.

1 Introduction

Cloud computing offers a large-scale distributed computing environment through a pool of abstracted, virtualized, dynamically-scalable and configurable computing resources. Unfortunately, due to unreliability in hardware or software, failure as the major obstacle to high service availability in cloud computing, is unavoidable. A fault tolerance feature provided by cloud vendors aims to overcome the impact of system failures and continue their functionality correctly even after the occurrence of failures, is needed. Currently, several fault tolerance models [8, 1, 6] are proposed generally involving the application developer to configure and operate cloud software based on cloud-specific features in order to run reliably. The major drawback and limitation of this type of approach is that requiries knowledge and experience from the developer in order to configure and integrate applications in an available fault-tolerance framework. This difficulty arises due

to (i) high complexity of the cloud platform, (ii) low available information about the underlying cloud infrastructure to its users. This results in intransparency and inflexibility of the Cloud architecture, and requiring too much effort by the application developer. Therefore, there is a demand for a reliable and automatic fault-tolerance management system without requirement for configuration and integration of applications by user. An efficient job distributor (load balancer) helps to remove critical conditions such as overload that causes a system failure and aims to improve system performance to make systems more reliable and fault-tolerant. Furthermore, as a part of a service layer, it brings more transparency in cloud infrastructures from a user's perspective. Recently, intelligent approaches have received attention for cloud job distribution and load balancing. Fuzzy theory [24], as a well-known artificial intelligence approach, has various characteristics that make it a suitable for control problems [12]. For us, it allows multiple possibly conflicting options – whether arising from an automated (machine) learning approach as multiple options or provided by different experts [3] – to be joined into a single decision that can be effectively enforced.

This paper proposes a fuzzy job distributor technique that ensures fault tolerance by properly distributing user job requests load among current available resources using anomaly and fault detection. By monitoring the current state of system and fairness in job distribution, we calculate the priority value for each resource and try to avoid overloading problems that are the cause of system failure. Upon detection of anomalies, the algorithm directs the system to apply a fault rejuvenation mechanism to an anomalously behaving virtual machine.

2 Fault Tolerance: Related Work and Positioning

Fault tolerance (FT) is the ability of a system to perform its function correctly even in the presence of internal faults. The purpose of fault tolerance is to increase the dependability of a system. Fault recovery mechanisms enable systems to correct the damaged state and restore to a known safe state after the system detects and verifies faults and anomalies leading to faults. Fault tolerance techniques can be classified into three main categories [6]: (i) redundancy techniques, (ii) load balancing strategies, and (iii) fault tolerance policies.

Redundancy is providing replication of system components such as hardware and software to provide more reliability in systems. Hardware redundancy techniques exploit additional hardware components. All redundant hardware executes the same task in parallel, and fault detection and masking can be achieved by majority voting techniques [18].

Load balancing fault tolerance strategies are based on improving fault tolerance based on load balancing performed using software models. In this case, a load dispatcher component distributes all incoming job requests among available resources. For example, Amazon EC2 uses elastic load balancing (ELB) to control how incoming requests are handled. Basically, in this context, it tries to *reduce* the likelihood of fault occurrences in the system by adequately distributing user job requests among available resources.

Fault tolerance policies can divided into *proactive* and *reactive* policies. The principle of proactive fault tolerance is to avoid recovery from errors and failure through preventative measures and proactively replace the suspected anomalous components by other working components. In contrast, reactive fault tolerance policies performs recovery from experienced failures.

Reactive Fault Tolerance is implemented in different ways. Firstly, *Checkpointing* records the system state periodically, allowing to restart the failed task from a recent checkpoint rather than from the beginning. Zhang et al. [26] propose a checkpointing strategy at user-level. The main drawback of this method is cost, which is significant in the case of large numbers of VM images in terms of storage space and restore processes. [25] proposes an asynchronous FT approach based on checkpointing by preserving data on surviving nodes to potentially accelerate recovering lost data with no overhead for checkpointing.

Secondly, *Replication* runs several task replicas on different resources. In the active model, all replicas receive the requests in the same order. In the passive model, one replica as the primary node receives the requests and all other replicas interact with the primary replica. To address reliability demands in PaaS cloud, a framework that automatically coordinates fault-tolerant applications based on the Byzantine fault-tolerant (BFT) protocol is proposed in [16]. In [19] an FT approach is proposed based on a checkpoint/replay technique for real-time computing to reduce the service time on the cloud infrastructure. Another reactive approach is *Job migration*, which migrates the failed task to another resource. *Task resubmission* is also widely used: the failed task is recommitted either to the same or a different resource.

Proactive Fault Tolerance can be distinguished into two important types: *Software Rejuvenation*: it immediately terminates an application and restarts it with a clean state at every rejuvenation interval [10]. *Pre-emptive Migration*: it counts on a feedback-loop control mechanism, i.e., constantly monitors and analyzes. It migrates the parts of an application that show anomalous behaviour and are likely to fail [7, 20]. In [17], a proactive coordinated FT (PCFT) approach based on particle swarm optimization (PSO) to minimizing the overall transmission overhead, overall network resource consumption is proposed. In [5], a VM placement model based on adaptive selection of fault-tolerant strategy for cloud applications is proposed. A predictive control approach for fault management in computing systems is presented in [14]. In most current clouds, (i) checkpointing, the process of recording and capturing recovery system state periodically during failure-free execution, and (ii) replication, the process of replicating tasks, are the most common fault tolerance strategies. The drawback of replication strategies is that they are rather expensive, i.e., higher cost for a device which contains multiple replicas. The advantage of checkpointing is that it does not require a high amount of hardware redundancy. However, the major drawback of checkpointing strategies is the time overhead of performing checkpoints.

Positioning of presented approach. Usually, the time overhead due to usage of fault tolerance policies may result in a negative impact on resource performance. In this work, in order to reduce the time overhead and improve the resource

utilization during the life cycle of system, we consider proactive fault tolerance strategies using load balancing as the central controller function [11, 13] and propose a fuzzy load balancer for fault tolerance management.

The proposed framework considers multiple objectives: (i) resource CPU utilization, (ii) fairness of distribution of job requests, and (iii) the history of fault rates for each resource. Our solution combines proactive techniques such as software rejuvenation with pre-emptive migration.

3 Fault Tolerance Management System

The first step of designing a fault tolerance mechanism as a service in cloud infrastructure is defining how the system works.

3.1 Self-Adaptive Anomaly and Fault Management Framework

Generally, the client jobs are deployed in VM instances. The fault tolerance properties of the system should be obtained through a core service that applies a coherent fault tolerance mechanism in a transparent manner. To this end, we define a fault tolerance controller as the fundamental module that monitors the current system state and enacts a fault tolerance mechanism. It allows us to control and handle hardware failure of user applications at the virtualization layer rather than for the application itself. The proposed fault tolerance approach is coded and run inside of it. Additionally, we use two more modules, namely *job distributor* and *anomaly/failure detector* components in our solution. The job distributor has the duty to distribute client job requests across a set of computing resources in resource pool based on current request load, priority and weight value for each resource. The anomaly and failure detector monitors resources to detect anomalies that might lead to failure and server crashes. A recovery mechanism can be applied after a failure is detected by this module. In this context, detection of node failures and application of the recovery mechanism are performed without requiring any changes to integrate a user application with fault tolerance approach.

Figure 1 shows the complete process of how the proposed fault tolerance system works. The fault tolerance controller gathers information from `ceilometer` and the current state of computing nodes in a resource pool. The `ceilometer` component provides telemetry services to collect metering data in OpenStack (which we use for implementation [2]). Then, the fault tolerance controller decides how to modify the priority and weight value of each node in the resource pool to reduce future anomalous behaviour. The job distributor distributes submitted jobs based on the weight value of each resource. During the life cycle of the system, the failure detector module detects anomaly and fault occurrences in the system and sends a recovery mechanism signal to the faulty node. Note that each module in Figure 1 has its own set of functional attributes.

Our anomaly detection framework aims to proactively prevent or detect faults: (i) detect anomalous undesirable performance degradation (as a concrete

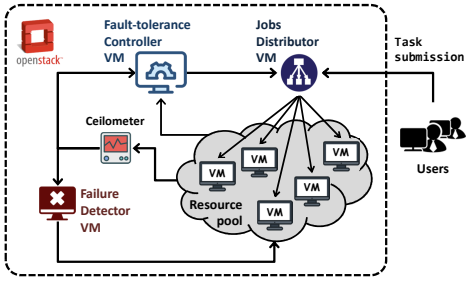


Fig. 1. Our fault tolerance framework

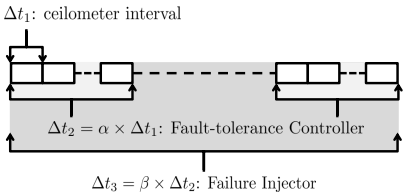


Fig. 2. interval check

anomaly) that might lead to failure, (ii) identify the symptoms and root causes of anomalous performance degradation to apply a proper corrective action, here using fuzzy job distribution, (iii) manage the relationships and dependencies between the symptoms, which are external manifestations of anomalous behaviour, and root causes, which are the reasons behind the performance degradation, and (iv) refine the future detection through applying a recovery mechanism on the identified faults and learning from the verified results to enhance the future fault detection and to continuously improve the deployment and the integration processes by using weight and priority adjustments. The following steps, aligned with the MAPE-K control loop framework [12], are carried out (see Figure 1):

- *M*onitoring: Anomaly/Failure Detection. This step collects data from the controller using `ceilometer`, structures this data to provide a sequence presentation that can be used to detect the obfuscated behaviour in data.
- *A*nalysis: Anomaly Identification and Diagnosis. To be able to identify and diagnose the fault root cause, we label the sequence representation in the anomaly detection step. The main points of that step are specifying the dependency and the relationships between faults, estimating the fault type (fault intensity level or the dispersal of anomaly within the managed resource) and distinguishing between fault (true anomaly diagnosing) and noise (false anomaly diagnosing). The distinction is specified based on assigning numerical values for each.
- *P*lanning and *E*xecution: Anomaly Recovery. After identifying and diagnosing faults, a recovery mechanism is applied to correct faults and remove their effects. The objective of fault removal is to isolate the affected component from the sequence presentation and delegate the incoming requests to another component or choosing an alternative solution to be used in the healing. This step is connected to the fault tolerance controller VM to reassign a new weight for the affected component(s) to be able to store the verified path(s) according to their new weight.

Furthermore, Recovery Validation evaluates the effectiveness of the previous steps in detecting faults, in which different types of faults can be considered (such as CPU-related fault, memory-related fault, disk-related fault and VM-related fault). The latency, throughput and response time are measured to infer

the performance of the measured components after faults isolation. The verified results are pushed back into the cloud (resource pool).

To gather status information from computing nodes in the resource pool, we use three different time windows during detection. Δt_1 specifies an interval after which the ceilometer component performs an update of the specified meter for the resource. Δt_2 is the sampling interval used by the fault tolerance controller machine, and the Δt_3 is used for sending periodic updates to the failure detector component. Generally, the time intervals Δt_2 and Δt_3 are proportional to the `ceilometer` interval parameter, i.e., Δt_1, in Fig. 2. For instance, if $\Delta t_1 = 10$ *seconds*, the value of Δt_2 and Δt_3 can be 10 *minutes* and 1 *hour*.

3.2 OpenStack

An important feature for users relates to the service uptime. To achieve high cloud availability and improve Service Level Objectives (SLOs) satisfaction, an efficient fault tolerance strategy needs to be employed. In contrast with a traditional manually configured approach, we propose an approach that used *active* and *runtime* monitoring for fault tolerance. It consists of several independent modules that work separately from each other in order to handle incoming job request load and perform fault tolerance in the target system.

In order to implement the fault tolerance controller and demonstrate its properties in an open IaaS solution, we have chosen the open-source OpenStack IaaS platform. It consists of components that control hardware pools of processing, storage, and networking resources throughout a data center. Users either manage it through a web-based dashboard, through command-line tools, or through a RESTful API. Fig. 3 shows the OpenStack core services. 1) Neutron is a system for managing networks and IP addresses; 2) Nova is the computing engine for deploying and managing virtual machines; 3) Glance supports discovery, registration and delivery for disk and server images; 4) `ceilometer` provides telemetry services to collect metering data; 5) Keystone provides user/service/endpoint authentication and authorization and 6) Heat is a service for orchestrating the infrastructure needed for cloud applications to run.

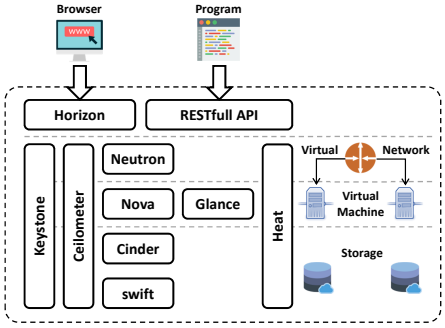


Fig. 3. An OpenStack block diagram

3.3 Job distributor strategies

Individual compute resources can easily suffer from heavy load or underload in the absence of a sufficient task dispatcher. The major cause for failure of the

process at the VM layer is, however, overloading. The job distributor strategies can be classified into two major categories: (i) *Static* approaches divide the load evenly among all available resources. They do not consider the current state of the system, which may lead to heavy system load or underload conditions. (ii) *Dynamic* approaches monitor the current state of the system for managing the load and aim for a more efficient load distribution. The main aim of a job distributor is to improve system performance by efficient usage of resources. The most common job dispatcher/controller strategies are:

- *Round-Robin (RR)*: In this strategy, as the name suggests, jobs are assigned to all servers in round-robin manner. *RR* does not consider factors such as the number of assigned job to the resource, CPU utilization, etc. Instead it treats all resources as equal and divides the traffic equally. It is the simplest strategy for implementation.
- *Weighted Round-Robin (WRR)*: It is an extension *RR* strategy where resources receive jobs according to their given weight value. Each resource can be assigned a weight. Resources with higher weights receive new job requests first compared to those with less weight, and resources with higher weights get more jobs than those with less weights.
- *Dynamic Weighted Round-Robin (DWRR)*: Since *RR* and *WRR* are static job distribution strategies and have to have knowledge of subsequent job requests, there are situations when already overloaded resources keep receiving more job requests although other idle resources are still available. By considering the real-time information and metrics of each resource such as current CPU utilization, *DWRR* applies dynamic weight assignment to avoid overloading and improves throughput of the whole system. The *DWRR* strategy reassigns a new weight value to the resources periodically.

3.4 Fuzzy Logic

Fuzzy logic [24] is an effective technique to describe complex systems with linguistic descriptions. A linguistic variable is a variable whose values are words in a natural language. For example, *"load"* is a linguistic variable, which can take the values as *"heavy"*, *"medium"*, *"light"* and so on. A Fuzzy Logic Systems (FLS) architecture consists of several components as shown in Figure 4: The *Fuzzification* module transforms the system inputs, which are crisp numbers, into fuzzy sets. The *Rules* (Knowledge Base) module stores `IF-THEN` rules provided by experts or learned from other sources. The *Inference* Engine simulates the human reasoning process by making fuzzy inference on the inputs and `IF-THEN` rules; the *Defuzzification* module transforms the fuzzy set obtained by the inference engine into a crisp value.

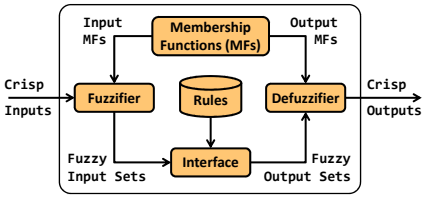
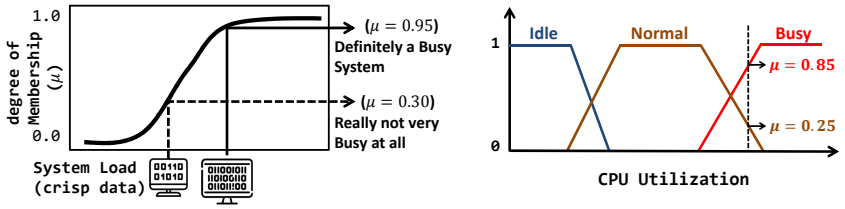


Fig. 4. Basic configuration of FLS

A membership function (MF) is a curve that defines how each point in the input space is mapped to a membership value (or degree of membership) between 0 and 1. MFs are used in the fuzzifier and defuzzifier modules of a FLS to map the non-fuzzy input values to fuzzy linguistic terms and vice versa.



(a) (b)

Fig. 5. Example of MFs

For example, Figure 5(a) shows a smoothly varying curve that passes from a *not loaded* system to *heavily loaded* system. The curve is known as a membership function (μ). Both systems are busy to some degree, but one is significantly less busy than the other. An important characteristic of fuzzy logic is that a value can belong to multiple sets at the same time. There are different forms of membership functions. For example, according to Figure 5(b), a CPU utilization value can be considered as *"normal"* and *"busy"* at the same time, with different degree of memberships. The most common types of membership functions are triangular, trapezoidal, and Gaussian shapes.
In a FLS, a rule base is constructed to control the output variable. Fuzzy rules are linguistic `IF-THEN` constructions that have the general form "`IF A THEN B`" where `A` and `B` are propositions contain linguistic variables. For instance, `IF` *load is high* `and` *target is medium* `THEN` *command is reduce.*

3.5 Fuzzy Fault Tolerance Management

Fuzzy control provides a solution to design a controller for a dynamic process based on available heuristic knowledge. Figure 1 earlier showed the general overview of our fault tolerance framework. Resulting from the `Resource pool` are the current weight and priority values for each available resource. Additionally, any changes of CPU utilization between two predefined intervals are collected from the `ceilometer`. The output of the fault tolerance controller is the modified weight value that determines whether the assigned job request for a resource should be increased or decreased in the next interval.

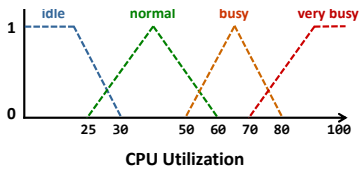


Fig. 6. Fuzzy membership functions for the input variable

According to current state, the change of the weight value between two intervals is calculated by the fuzzy controller and send to the job distributor module as the adaptive weight value for the resource to be used for the next interval. Based on the change of CPU utilization and loaded job request to the resource

in the previous interval, the fuzzy fault tolerance controller determines the new value for weight and priority of each available resource for the next interval.

As it described before, the fuzzifier and defuzzifier modules (Figure 4) in the fuzzy controller internally work with linguistic variables and values. The input numeric values are measured and converted to the corresponding linguistic values by the fuzzification module, and the reverse operation is performed by the defuzzification module.

Based on the linguistic input value, the interface module selects the appropriate rule to be applied and produces the linguistic output value. Both fuzzifier and defuzzifier use an MF to convert numeric values to linguistic values and vice versa. The MF maps each numerical value to a membership value (certainty level) between 0 and 1 (0 completely uncertain, 1 completely certain). Figure 6 represents our membership function, where the x-axis represents CPU utilization values and the y-axis membership values. Based on possible levels of CPU utilization, which is the metric that represents how busy a processor core is, in this work, the linguistic variables representing the value of resource utilization level are divided into four levels: *idle*, *normal*, *busy* and *very busy*. To determine the boundary values of each linguistic variable, we collected the required data from several experts in cloud application management, and used the average of all the responses for each variable.

Our fuzzy fault tolerance controller uses the following anomaly identification rules that help in recognising possible failure and that result in job distribution and weight/priority adjustment as the response:

- A resources is defined as *overloaded* if its CPU utilization exceeds a given threshold for a predefined time frame. In this situation, the fuzzy controller determines the appropriate values of load weight and priority parameters for the target resource according to its current level of CPU usage. By adjusting the weight value, the job distributor will send less job requests to this resource until its CPU usage is in a safe mode.
- An *underload* situation occurs whenever the CPU usage of the resource becomes low value for a given time window, i.e., the resource has a low number of jobs to execute and mostly is in idle mode. In this case, the fuzzy controller modifies and increases the weight and priority value of idle resources to receive more job requests from job distributor, thus reducing likely failure elsewhere on other nodes.

Anomaly management happens in the following two ways. Firstly, overloading is an anomaly taken as an indication that failure is likely to happen, i.e., performance degradation is a root cause for failures, and underload is an anomaly that signals an opportunity to reduce likely failure elsewhere by allocating load to the current node. Secondly, a further hypothesis of the anomaly framework is that incorrect weight and priority negatively impacts on fault occurrences. The incoming job load to each resource are determined based on its weight and priority values. Therefore, in order to have a fair distribution on user job requests and avoid of over/under load situations, our fuzzy controller has duty to modify these parameters based on loaded job request to the resource, the history of

Technique	Strategy	Weight Value
Equal weighted job distributor ($Equal\text{-}W$)	Resources receive job requests in a circular fashion without considering resource metric such as CPU utilization and fault tolerance, i.e., all resources have same weight value (W)	$\forall\, r_i, r_j \in RP \mid W(r_j) = W(r_i)$
least-CPU utilization weighted job distributor ($cpu_{util}\text{-}W$)	Resources are weighted based on their CPU utilization, and job requests are distributed in proportion to the weight value. Higher values will be assigned to the resource with lower CPU utilization	$\forall\, r_j \in RP \mid W(r_j) = 100 - cpu_{util}(r_j, \Delta t_2)$
Fuzzy weighted job distributor ($Fuzzy\text{-}W$)	Resource weight value is obtained by the fuzzy fault tolerance controller based on the current CPU utilization and the history of weight value for the resource	$\forall\, r_j \in RP \mid W(r_j) = Fuzzy(r_j, \Delta t_2)$

(1) Resource Pool contains of available resources.
(2) Average CPU utilization of resource r_j during previous time window Δt_2.
(3) Weight value of resource r_j based on CPU utilization during previous time window Δt_2.

Table 1. Description of compared strategies used in the controller evaluation

fault rates and the change of CPU utilization for target resource. In this way, a proactive pre-emptive migration FT strategy is applied.

4 Implementation

We implemented a prototype of the proposed fuzzy logic fault tolerance controller in OpenStack. The Fuzzy Fault Tolerance controller is a based on a fuzzy logic-based feedback control loop. It continuously monitors the resource utilization (using `ceilometer`) and triggers the controller at each interval check period. According to the utilization values for each available resources, the fuzzy controller module identifies appropriate load weight values in anomalous situations.

In our implementation, we assume one or more VM instances as members in the `Resource pool`. We use a minimal Linux distribution, namely the `cirros` image that was specifically designed for use as a test image on cloud platforms such as OpenStack. Each instance (VM) receives a job request and executes it. In our experiment, we consider all job requests submitted by different users as a CPU bounded type. In order to control and manage weight values of available resources by a fuzzy logic controller, we added an additional VM resource, which acts as a fault tolerance controller and decides and reassigns weight values periodically. For the fault tolerance controller, due the impossibility of installing any additional package in the `cirros` image, we considered a VM machine running Linux Ubuntu-based images. Figure 7 illustrates the implemented system in OpenStack. The created job distributor distributes user job requests across a set of resources, i.e., the `Resource pool`. The strategies used in the job distributor controller VM for evaluation (a comparison between our proposed fuzzy controller and two other traditional approaches) are summarized in Table 1.

Figure 7 shows the complete process of the proposed fuzzy fault tolerance approach. First, the fault tolerance controller gathers information from the job distributor, ceilometer and the current state of members (available resources) in

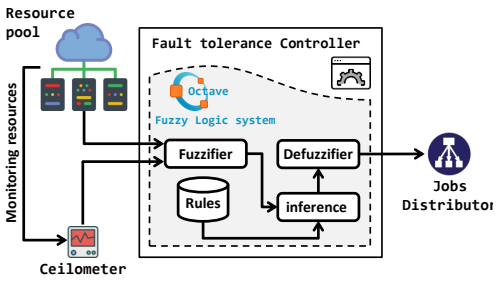


Fig. 7. Overview of the implemented Fault Tolerance Controller

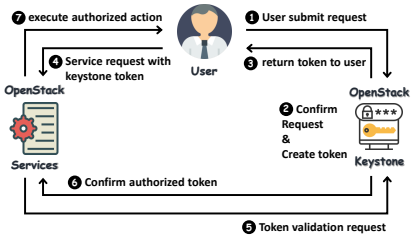


Fig. 8. `cURL` process of calling the OpenStack API

the resource pool, then identifies appropriate load weight value for a resource according to the situation in order to adjust anomalous situations. For example, if a resource is overloaded, the controller determines that the incoming job load to the resource should be decreased, therefore it reassigns a new weight value for the resource to reduce the submitted job requests. The proposed fuzzy logic controller is coded and run inside of the fault tolerance controller machine.

For some parameters in the proposed algorithm, such as the current number of VM instances or workload, we need to call the OpenStack API. For example, the command `nova list` shows a list of running instances. The API is a RESTful interface, which allows us to send URL requests to the service manager to execute commands. Due to the unavailability of direct access to the OpenStack API inside of the fault tolerance controller machine, we used the popular command line utility `cURL` to interact with a couple of OpenStack APIs. `cURL` lets us transmit and receive HTTP requests and responses from the command line or a shell script, which enabled us to work with the OpenStack API directly.

In Figure 8, the process of using `cURL` to call OpenStack APIs is shown. First, we send a request authentication token by passing credentials (username and password) from OpenStack Identity service. After receiving `Auth-Token` from Keystone, the user can combine the authentication token and Computing Service API Endpoint to send a HTTP request and receive the output. We use inside the fault tolerance controller machine to execute OpenStack APIs and collect required outputs. By combining these settings, we are able to run the fuzzy logic approach as the controller of fault tolerance management in OpenStack.

5 Experimental Comparison

The evaluation aims at showig the effectiveness of our fuzzy logic controller for fault tolerance management in comparison to other job distribution strategies.

5.1 Experimental setup and benchmark

In our experiment, the proposed fuzzy logic approach was implemented as full working systems and was tested in the OpenStack platform. The number of

available resources considered in our experiment was set to 4 VMs. The term job workload refers to the user request arrival. Job workload is defined as the sequence of users submitting the job request that needs to be handled by the job distributor. To evaluate our proposed approach, we considered a multiple number of workloads. In each workload scenarios, there are a set of job requests submitted by individual users. Each job request submitted by a user is considered as a CPU bounded job. At each workload scenario, the duration of job execution was set by *Poisson Distribution*. Several workload scenarios were executed and the total duration of our experiment was 2 weeks.

In order to evaluate the proposed approach and generate/manage faults in the target system, we used a fault detector VM, shown in Figure 1, as a single system fault model. By gathering information from the ceilometer about the current situation of each available VM, the fault detector is able to detect whether the resource goes into an anomalous state (over/underload) or not. Based on current CPU utilization of the resource in the defined time window, the fault detector module detects if a target resource is overloaded for a period, and sends a recovery signal to the target resource. To simplify the fault recovery process here, we consider hardware rejuvenation as the recovery fault tolerance strategy.

Additionally, we compared the proposed fuzzy fault tolerance approach with two other algorithms, namely $Equal\text{-}W$ and $cpu_{util}\text{-}W$, as shown in Table 1. In the $Equal\text{-}W$ approach, each available resource receives job requests in a circular fashion without considering resource metrics such as CPU utilization and fault tolerance, i.e., all resources have the same weight value (W). In contrast, the $cpu_{util}\text{-}W$ approach, by monitoring resource CPU utilization, the weight values are assigned dynamically, and job requests are distributed in proportion to the weight value. There is other research on load balancing strategies [15, 23, 4], which aims to improve objectives such as resource response time, which are similar in terms of the monitoring set up, but not the configuration of the analyses and enactment strategies for fault tolerance.

5.2 Comparison metrics

We measure the performance of the cloud environment during the whole period for each executed scenario. The metrics used for comparison are:

- *CPU utilization*: as a key metric considered in resource management across clouds, it is a function of time and is denoted by the amount of time a CPU is busy for handling work during a specific interval. It is reported as a percentage. CPU anomalies appear if its utilization goes beyond a high threshold (e.g., 80%) for a sustained period of time.
- *Failure rate*: is the representation of the total number of failures experienced during the experiment for each scenario. It widely used to represent the stability and reliability of a target system.

5.3 Results and discussion

Figure 9 shows the distribution of the CPU utilization metric (`cpu_util`) obtained by comparing the algorithms during our experiment for each individual available resource. For all VMs, our approach (*Fuzzy-W*) obtained a better distribution range, with cpu_{util}-*W* consistently second best, followed by *Equal-W* as last. The wider range of CPU usage distribution shows that the job request load has a more fair distribution among all available resources. Fairness is defined based on the CPU usage of each resource and tries to avoid CPU overloading for a long period. In this context, fairness represents the quality of service provided by a cloud service and it tries to avoid SLA (Service Level agreement) violation due to host overloading. By using dynamic weight and priority values for load job request distribution, both *Fuzzy-W* and cpu_{util}-*W* algorithms try to overcome the overloading anomaly situation that causes system failures.

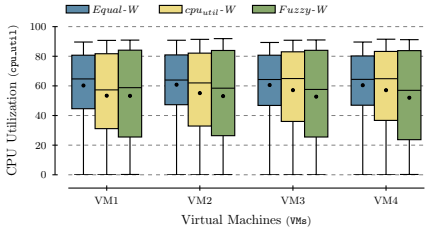


Fig. 9. CPU utilization (`cpu_util`)

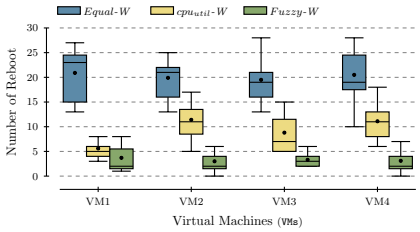
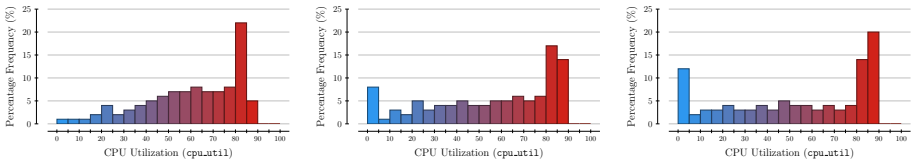


Fig. 10. Failure Rates

In figures 11(a), 11(b), and 11(c), the bars represent the percentage frequency of CPU utilization among all available resources for the compared algorithms, i.e., *Equal-W*, cpu_{util}-*W* and *Fuzzy-W*, respectively.



(a) *Equal-W* (b) cpu_{util}-*W* (c) *Fuzzy-W*

Fig. 11. Percentage frequency of CPU utilization

5.4 Comparison of effectiveness

Figure 10 shows the distribution of reboot occurrences (resulting from failures) for individual resources during of our experiment under several workload scenarios. As it mentioned before, both *Fuzzy-W* and cpu_{util}-*W* approaches have better CPU usage distribution compared to *Equal-W* (Figure 11). However, due to a higher distribution of CPU utilization in *Fuzzy-W*, at each time interval for the failure detector, we have lower average values for CPU utilization, and it shows a significant reduction of the number of reboot occurrences.

6 Conclusion

We have proposed a new fuzzy logic-based load balancer for fault tolerance in IaaS cloud platforms. The proposed approach employs a fuzzy logic strategy to assign a weight and priority value to each available resource as a proactive strategy in anomalous situations. By monitoring the current state of a system, it tries to adjust the weight value for each resource in order to achieve: (i) fairness job distribution, (ii) avoid anomalous situations such as overloading that causes a system failure, and (iii) improve throughput of the whole system. Overloading of a system may lead to poor performance which can increase failure rates and SLA violation. Underload is also dealt with to reduce anomalies elsewhere.

The assignment mechanism for choosing the appropriate weight value in the proposed approach is based on a fuzzy logic system (FLS) and collected metering data as its input. By considering the real-time information and collected metrics of each resource, it achieves a more efficient load distribution and reduces the occurrence of failures in the system. The proposed approach was coded and implemented in OpenStack, an open-source IaaS platform, to demonstrate the practical effectiveness of proposed approach, and evaluated based on important metrics, including distribution of CPU utilization and failure rate during of our experiment for each individual resource. The experimental results revealed that using a fuzzy approach the proposed approach outperformed the other strategies considering all the above mentioned metrics, especially in failure rate parameters, which is the main objective here.

We plan to apply the solution also to container-based virtualisation [21, 9] towards an edge-cloud management platform [22] in the future.

Acknowledgement. This work was partly supported by IC4 (the Irish Centre for Cloud Computing and Commerce), funded by EI and the IDA.

References

1. Z Amin, H Singh, and N Sethi. Review on fault tolerance techniques in cloud computing. *Intl Journal of Computer Applications*, 116(18), 2015.
2. H Arabnejad, P Jamshidi, G Estrada, N El Ioini, and C Pahl. An auto-scaling cloud controller using fuzzy q-learning-implementation in openstack. In *European Conference on Service-Oriented and Cloud Computing*, 2016.
3. H Arabnejad, C Pahl, P Jamshidi, and G Estrada. A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling. In *Intl Symp on Cluster, Cloud and Grid Computing CCGrid*, 2017.
4. Z Chaczko, V Mahadevan, S Aslanzadeh, and C Mcdermid. Availability and load balancing in cloud computing. In *Intl Conf on Comp and Softw Modeling*, 2011.
5. X Chen and JH Jiang. A method of virtual machine placement for fault-tolerant cloud applications. *Intel. Automation & Soft Computing*, 22(4):587–597, 2016.
6. MN Cheraghlou, A Khadem-Zadeh, and M Haghparast. A survey of fault tolerance architecture in cloud computing. *Jrnl of Netw & Comp Appl*, 61:81–92, 2016.
7. C Engelmann, GR Vallee, T Naughton, and SL Scott. Proactive fault tolerance using preemptive migration. In *Intl Conf on Par, Distr and Netw-based Proc*, 2009.

8. A Ganesh, M Sandhya, and S Shankar. A study on fault tolerance methods in cloud computing. In *Intl Advance Computing Conference*, pages 844–849, 2014.
9. R Heinrich, A van Hoorn, H Knoche, F Li, LE Lwakatare, C Pahl, S Schulte, and J Wettinger. Performance engineering for microservices: Research challenges and directions. In *ACM Intl Conf on Performance Engineering Companion*, 2017.
10. Y Huang, C Kintala, N Kolettis, and NS Fulton. Software rejuvenation: Analysis, module and applications. In *Intl Symp Fault-Tolerant Computing*, 1995.
11. P Jamshidi, C Pahl, and NC Mendonça. Managing uncertainty in autonomic cloud elasticity controllers. *IEEE Cloud Computing*, 3(3):50–60, 2016.
12. P Jamshidi, A Sharifloo, C Pahl, H Arabnejad, A Metzger, and G Estrada. Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. In *ACM Intl Conf on Quality of Software Architectures (QoSA)*, pages 70–79, 2016.
13. P Jamshidi, AM Sharifloo, C Pahl, A Metzger, and G Estrada. Self-learning cloud controllers: Fuzzy q-learning for knowledge evolution. In *Cloud and Autonomic Computing (ICCAC), 2015 International Conference on*, 2015.
14. R Jia, S Abdelwahed, and A Erradi. A predictive control approach for fault management of computing systems. *Perf Evaluation Review*, 43(3):16–20, 2015.
15. NJ Kansal and I Chana. Cloud load balancing techniques: A step towards green computing. *Intl Jrnl of Computer Science Issues*, 9(1):238–246, 2012.
16. B Li and R Kapitza. Bft-dep: Automatic deployment of byzantine fault-tolerant services in paas cloud. In *Distributed Appl and Interoperable Syst*, 2016.
17. J Liu, S Wang, A Zhou, S Kumar, F Yang, and R Buyya. Using proactive fault-tolerance approach to enhance cloud service reliability. *IEEE TCC*, 2016.
18. RE Lyons and W Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Jrnl of Research & Development*, 6(2), 1962.
19. B Mohammed, M Kiran, KM Maiyama, MM Kamala, and I-U Awan. Failover strategy for fault tolerance in cloud computing environment. *Software: Practice and Experience*, 2017.
20. AB Nagarajan, F Mueller, C Engelmann, and SL Scott. Proactive fault tolerance for hpc with xen virtualization. In *Intl Conf on Supercomp*, 2007.
21. C Pahl, A Brogi, J Soldani, and P Jamshidi. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing*, 2017.
22. C Pahl, S Helmer, L Miori, J Sanin, and B Lee. A container-based edge cloud paas architecture based on raspberry pi clusters. In *IEEE Intl Conf on Future Internet of Things and Cloud Workshops (FiCloudW)*, 2016.
23. M Randles, D Lamb, and A Taleb-Bendiab. A comparative study into distributed load balancing algorithms for cloud computing. In *AINA Workshops*, 2010.
24. P Vas. *Artificial-intelligence-based electrical machines and drives: application of fuzzy, neural, fuzzy-neural, and genetic-algorithm-based techniques*. OUP, 1999.
25. Z Wang, L Gao, Y Gu, Y Bao, and G Yu. A fault-tolerant framework for asynchronous iterative computations in cloud environments. In *ACM Symp on Cloud Computing*, pages 71–83, 2016.
26. Y Zhang, D Wong, and W Zheng. User-level checkpoint and recovery for lam/mpi. *Operating Systems Review*, 39(3):72–81, 2005.