A New Fault-Tolerant Algorithm Based on Replication and Preemptive Migration in Cloud Computing

Abderraziq Semmoud[*] and Badr Benmammar[†]
Faculty of science, Department of Computer Science,
Abou Bakr Belkaid University, Tlemcen, Algeria

Mourad Hakem[‡] and Jean-Claude Charr[§]
DISC Laboratory, Femto-ST Institute, UMR CNRS, Université de Franche-Comté, France

(Dated: September 11, 2022)

Cloud computing is a promising paradigm that provides users higher computation advantages in terms of cost, flexibility, and availability. Nevertheless, with potentially thousands of connected machines, faults become more frequent. Consequently, fault-tolerant load balancing becomes necessary in order to optimize resources utilization while ensuring the reliability of the system. Common fault tolerance techniques in cloud computing have been proposed in the literature. However, they suffer from several shortcomings: some fault tolerance techniques use checkpoint-recovery which increases the average waiting time and thus the mean response time. While other models rely on task replication which reduces the cloud's efficiency in terms of resource utilization under variable loads. To address these deficiencies, an efficient and adaptive fault tolerant algorithm for load balancing is proposed. Based on the CloudSim simulator, some series of test-bed scenarios are considered to assess the behavior of the proposed algorithm.

I. INTRODUCTION

The cloud is emerging as a wide-scale distributed computing infrastructure that enables resource sharing and coordinated problem solving in today's world that needs information anywhere and anytime. It provides highly scalable, secure and efficient mechanisms for discovering and negotiating remote access to computing resources in a transparent manner.

System load is a measure of the amount of work that a computer system performs. If the load on some computers is generally heavier than on others, or if some processors execute tasks more slowly than others because of resources heterogeneity, they will be overloaded. Load balancing aims to ensure that all processors share the workload over the long term. Even though load balancing is essential to ensure high availability of applications in an increasingly critical environment, failures become inevitable as the number of components in the cloud system increases. Therefore, a load balancing algorithm should have the fault tolerance capability, i.e., it should perform uniform load balancing despite the presence of arbitrary node or link failures. One of two approaches can be adopted to provide fault tolerance: Proactive or Reactive Fault Tolerant Policy. The principle of the former is to avoid failures by predicting them and proactively taking preventive actions before a failure occurs. Preemptive migration [1], software rejuvenating [2], self-healing [3] are some proactive fault tolerant techniques. While the reactive one follows some kind of policies and helps to recover from failed state when a failure occurs. There are two classes of reactive fault tolerance techniques. The first class lets the application continue the execution until its termination even if some nodes fail. Compensation mechanisms in the run-time environment or in the application algorithm avoid the complete failure of the whole application [4]. This class of methods includes Replication [5], Algorithmic-based fault tolerance [6], Natural Fault-tolerance [7], Rescue work-flow [8] and Failure masking [9]. While in the second class, the effect of failures is repaired either by continuing the execution considering that the application will recover later a normal state or by re-executing the failing parts of the application. Rollback-Recovery [10], Forward-Recovery [11], Task re-submission [12], Retry [13], task migration [14] are common techniques of this class.

In this work, we design a hybrid failure management mechanism for load balancing in a cloud computing environment. This mechanism is based on preemptive migration and replication to proactively take preventive actions before a failure occurs and reactively manage the occurrence of failures. With this mechanism, we can effectively reduce resource usage, costs, and network traffic in data centers.

In the following, we summarize the contributions and the novelties of the presented study:

1. The design of a new distributed algorithm for fault-tolerant load balancing which is able to deal with scalability and dynamicity of the targeted system.

2. The combination of proactive and reactive fault tolerance techniques in an adaptive way to cope with unpredictable failures.

3. Ensuring fault tolerance while maintaining the load of the system balanced as much as possible.

[*] abderrazak.semmoud@univ-tlemcen.dz
[†] Badr.Benmammar@univ-tlemcen.dz
[‡] mourad.hakem@univ-fcomte.fr
[§] jean-claude.charr@univ-fcomte.fr

The rest of this paper is organized as follows: Section II describes some related works on load balancing and fault tolerance techniques for the cloud environment. The system modeling and the problem formulation are given in Section III. In Section IV, the distributed fault tolerance approach is presented. Section V focuses on the simulation setup and experimental results. Finally, the article ends with a summary of the contributions and some future work.

II. RELATED WORKS

In this section, the relevant techniques proposed in the literature to deal with the fault tolerance management are reviewed.

In [15], an overlay architecture for large-scale data networks is presented. The architecture named Saturn is maintained over Distributed Hash Tables (DHTs) that processes queries in an efficient manner and ensures access load balancing and fault-tolerance. Placing consecutive data values in neighboring peers speeds up query processing. However, such a placement is very sensitive to load imbalances. A new order-preserving multiple-ring architecture is used by Saturn to deal with failures. The use of a new order-preserving hash function ensures fast processing of range requests. Replication across and within data rings ensure query load balancing and fault tolerance, respectively. Wang and al. [16] have proposed a protocol called the Dual Agreement Protocol of Cloud-Computing (DAPCC). It divides the topology into two layers named A and B. Layer A has some nodes which are directly related to customers and layer B has set of nodes of the same type which are directly attached with the nodes of layer A. DAPCC achieves agreement among all nodes on a common value in a minimal number of message exchanges and can tolerate a maximum number of allowable failed components.

In [17], Chang and al. have presented a fault-tolerant load balancing scheme for current web services. In this work, each block within a file is replicated at least three times depending on the access frequency. To reduce the likelihood of data loss, each of the replicas is located on a different server. When the control center detects that the number of replicas of a block is less than three due to a network problem or a hardware failure, the control server starts immediately a new replication of the block. Fang and al. [18] combine the fault tolerance and workload balancing mechanisms in the Distributed Stream Processing Engine (DSPE) to reduce the overall resource consumption while preserving the system's interactivity, high-throughput, scalability and high availability. Based on a data level replication strategy, the proposed method can handle a dynamic scenario of data asymmetry and node failures. During the fluctuation of the incoming flow distribution, the workload is rebalanced by selectively deactivating the data in overloaded nodes and activate their replicas on underloaded ones to minimize load's migration within the stateful operator; when a failure occurs, the system activates the replicas of the affected data to ensure rapid recovery after a failed processing task while keeping the workload balanced.

The proposed work, in [19], introduces an innovative perspective on adopting a fault tolerant mechanism that shades the cloud server implementation with cloud selection to avoid network congestion and health monitoring. Since it can preserve the system data availability, the cloud selection is induced to prevent the network traffic. With the proposed framework, a proactive fault tolerance technique is provided and the experimental study revealed reduced overhead and energy consumption. Nirmala et al. [20] propose a new fault tolerant workflow scheduling algorithm for large scale scientific workflow applications that learns replication heuristics in an unsupervised manner. Authors propose a light weight synchronized checkpointing method to identify failed task and estimate replication count using heuristics. Different machine learning algorithms were implemented to predict failures for different type of workflow applications. In the work of Chatterjee et al. [21], a two-level fault tolerant load balancing algorithm called GBFTLB (Gossip-Based Fault-Tolerant Load Balancing) has been proposed. It considers that processors are heterogeneous and uses processor capacity to allocate the load to it. A new communication model is designed for global communication between processors in order to manage the loss of connectivity and minimize communication costs. GBFTLB can operate in asynchronous and synchronous networks and can be applied to systems with arbitrary topologies. The GBFTLB aims to optimize the number of rounds and the number of messages required in fault tolerant load balancing.

To achieve low fault tolerance overhead and fast failure recovery, a new approach based on replication technique for graph computation using either edge-cut or vertex-cut is proposed in [6]. The presented technique which is called Imitator leverages existing vertex replication to tolerate node failures, by extending existing graph-parallel computations. It provides two recovery techniques: Rebirth and Migration based recovery. The experimental results showed that Imitator has good performances in terms of low execution overhead, and fast crash recovery. In [22], a proactive technique which relies on the Central Processing Unit (CPU) temperature is introduced to deal with node failures within a virtualized cloud federation environment. First, the authors formulate the studied fault tolerance problem as an Integer Linear Programming (ILP) multiobjective model to optimize profit, migration cost, and resources redistribution in case of faults. Then, to determine the best cloud service providers from a reserve of existing ones, a domination and preference relationships-based scheme is designed. The algorithm's performance has been evacuated on a real cloud dataset, and the obtained results reveal that migration cost has a noticeable impact on the achieved overall profit within the cloud.

To minimize the communication cost induced by fault tolerance and deal with dynamic distributed platforms, the authors, in [23], propose a new collaborative checkpoint-rollback recovery scheme which takes partial snapshot of the system by using appropriate coordination among the network's nodes. They demonstrate throw simulations, that the proposed approach is able to decrease the number of exchanged messages of the coordination process. A new Fault-tolerant and real time algorithm which is called ReadyFS is proposed in [24] for scheduling scientific workflows in a cloud. The proposed approach seeks to improve resource utilization while guaranteeing fault tolerance and deadline requirements. Both replication and check-pointing with delay execution are investigated, and the conducted comparison with other existing techniques using real-world workflow applications showed the usefulness of the authors' method in terms of fault tolerance efficiency and resources management.

In recent work [25], an adaptive approach has been proposed to deal with the problem of fault tolerance in cloud computing. In this approach, a fuzzy-based method is used to detect failures, and a predictive approach is implemented to monitor the provided system. In order to increase fault tolerance and load balancing when failures occur, the checkpointing method is used which reduce the time as well as the processing cost of task migration. To detect failures, a fuzzy system with throughput, response time and workload as input parameters was designed. The scalability and computational overhead of the proposed approach have not been evaluated. To reduce the effect of transient failures, the authors of [26] proposed an aggressive fault tolerance approach in a cloud environment to detect and recover from failures. An intelligent decision agent is used by the aggressive fault detection and recovery module to detect and recover from faults. The intelligent decision agent makes decisions on different types of software, hardware, and communication faults. It improves the performance of fault tolerance schemes and reduces complexity compared to other existing techniques such as replication, resubmission, and checkpointing techniques.

As reported above, unlike earlier works, this study investigates the combination of proactive and reactive fault tolerance techniques to deal with unexpected failures during the load balancing process. The objective is to take advantage of each technique in adaptive way to improve the overall performances in terms of system reliability and Quality of Service (QoS) seen by end users. To address this, we make use of Preemptive Migration and Replication in order to proactively take preventive actions before a failure occurs and then reactively avoid the impact of Virtual Machine (VM) failure. Moreover, the replication process is performed in a balanced way as we can use replicas to smooth fairly the load in the system.

III. SYSTEM MODELING

Let $G(t) = (VM(t), E(t))$ be a dynamic graph representing the mapping between the VMs. The graph is dynamic in the sense that at every unit of time t, some VMs can enter and leave the system. $VM(t)$ is the set of virtual machines in the system at time t. $E(t)$ is the set of bidirectional links between VMs such as VM_i and VM_j can send messages to each other at time t if and only if $(VM_j, VM_j)E(t)$. The system is considered failure-prone where nodes and/or links may fail with a mean time to failure of mttf. $N_i(t)$ is the set of VMi's direct neighbors at time t. Each VM_i has a set CR_i of cores. Let $ST_i(t)$ be the set of tasks submitted to VMi at time t. The overall size of the tasks $SZ_i(t)$ assigned to VM_i at time t is defined as follows:

$$SZ_i(t) = \sum_{task \in ST_i(t)} SIZE(task) \tag{1}$$

Let CT_{ik} the completion time of a task T_k in VM_i. The waiting time of the task Tk is defined as a function of its arrival time AT_{ik} at VM_i and the completion time of task T_{k-1}:

$$WT_{ik} = Max\{CT_{i(k-1)}, AT_{ik}\} - AT_{ik} \tag{2}$$

System response time is the average time between submitting a task to the cloud and its completion.

$$RT = Average\{WT_{ik} + ET_{ik}\}, i = 1...m, k = 1...n_i \tag{3}$$

Makespan is the overall completion time of the tasks in the set T. It is defined as the following:

$$MS = Max\{CT_{ik}\}, i = 1...m, k = 1...n_i \tag{4}$$

where n_i is the number of tasks executed by VM_i. Let $REP_{ik}(t)$ be the set of VMs receiving the replicas of task T_k from VM_i at time t. The reliability of VM_i at time t is denoted by $REL_i(t)$.

The main objective of the proposed fault-tolerant algorithm is to identify the suitable replication nodes and reduce the number of replications while maximizing both system's reliability and tasks' completion rate.

IV. THE PROPOSED ALGORITHM

The presented study is essentially an extended version of the STLB (Starvation Threshold Based Load Balancing) algorithm of [27], which was designed to tackle the problem of load balancing in cloud computing environments. It differs from the initial version in the way that it takes the fault tolerance requirement into account by combining both proactive and reactive techniques in an adaptive way. The proposed RPMFT (Replication and Preemptive Migration based Fault Tolerance) algorithm adds a powerful part to the initial version by mixing

Preemptive Migration and Replication in order to proactively take preventive actions before a failure occurs and reactively avoid the effects of VM's failure. In addition, the replication is done in balanced way as we can use task replicas to balance the load in the system. The main goal of the proposed approach is to ensure fault tolerance while:

- Reducing the number of replications using:
  - A hybridization of proactive and reactive approaches;
  - A reliability factor that depends on an adaptive confidence index.
- Choose the suitable replication nodes based on neighbor's load.

A. Preemptive Migration

The feedback loop control mechanism is an essential component of proactive fault tolerance using preemptive migration (Figure 1), where application is constantly monitored and analyzed as preventative actions can be taken to avoid imminent application failure by preemptively migrating parts of an application (process, task, or virtual machine) away from nodes that may fail.

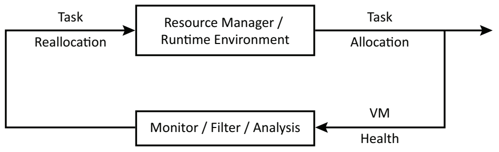


FIG. 1: Feedback-loop control.

The feedback loop (Figure 1) consists of continuous application health monitoring and reallocation of application parts. Application health monitoring encompasses hardware and software monitoring. The reliability factor $REL_i(t)$ represents the health status of VMi. As shown in Figure 2.(b), once it is detected that the reliability factor does not meet an adaptive threshold called confidence index $CI_i(t)$, the VM's tasks are reallocated to other VMs. Reallocation evicts tasks from one or more nodes and excludes them from further use, thereby reducing the number of replicated tasks and thus resource utilization. Tasks are migrated to the most underloaded neighbor to balance the load among the nodes in the system.

B. Replication

Replication based technique is one of the popular approaches to ensure fault tolerance. Two major classes of replication techniques exist: active and passive replication. In the case of active replication, all replicas play the same role: there is no centralized control and all replicas start in the same initial state and execute the same operations in the same order. Each will replica will perform the same work and produce the same output. Active replication is failure transparent. However, it requires many processing resources. The second type called passive replication uses one replica, called the primary, for a special role: all operations requested by clients are directed to it. The primary is also responsible to execute the operations, update the other replicas and respond to the client. In this work, a hybrid-based approach is designed by involving characteristics of both active and passive replication. In this technique, one replica processes the requests received from clients. However, another replica handles the client request when the VM containing the primary replica becomes unreliable.

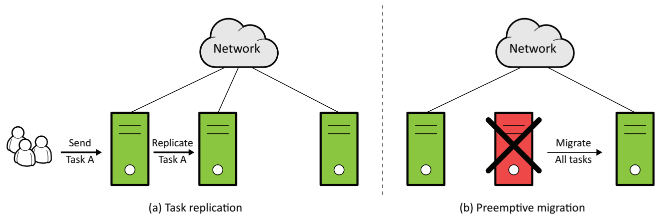


FIG. 2: The RPMFT architecture.

C. The fault tolerant algorithm

The proposed model handles faults that may occur in all available virtual machines. It tolerates faults based on reliability and load assessment to reschedule tasks to the most reliable virtual machines while preserving load balancing. As described in Algorithm 1, once a new task is received, it is replicated to the least loaded neighboring node (Figure 2.(a)) that meets the following condition:

1. $(VM_j \in N_i(T)) \land (FL_j(t) = 1)$
   $\land (L_j(t) \leq min(L_m(t)|VM_m \in N_i(t)))$

In order to have at least two replicas on reliable VMs, the reliability factor is evaluated and a second replication is performed if the receiving VM is not reliable. Moreover, to take preventive actions before a failure occurs. The health of the VM is continuously monitored and analyzed. Recall that, as defined above, the reliability factor $REL_i(t)$ represents the health state of VM_i as a function of fan speed, processor temperature, and average processor utilization. Once it is detected that the reliability factor does not meet the confidence index, the VM's tasks are replicated to other VMs.

When we deal with fault tolerance, a policy must be defined to identify when to take actions (proactively or

Algorithm 1 RPMFT

```
1:  while running do
2:      if receive new task $Tk$ then
3:          $REP_ik(t) = \emptyset$
4:          Replicate $T_k$ on $VM_j$ satisfying condition 1
5:          $REP_{ik}(t) = REP_{ik}(t) \cup VM_j$
6:      end if
7:      if $REL_i(t) < CI_i(t)$ then
8:          for $T_m \in VM_i$ tasks set do
9:              if $|REP_{im}(t)| = 0$ then
10:                 Migrate $T_m$ on $VM_j$ satisfying condition 1
11:             end if
12:         end for
13:     end if
14: end while
```

reactively) to ensure the system's reliability and scalability. In this work, a reliability factor $REL_i(t)$ is used to achieve this goal. This factor is calculated based on hardware and software satus as well as the history of the machine state. Indeed, a VM relocates tasks to its neighbors only if its reliability factor is lower than a confidence index threshold $CI_i(t)$.

V. PERFORMANCE EVALUATION

To evaluate the performance of the proposed algorithm, series of experiments were conducted on CloudSim [28] which is a cloud simulator. The study is performed on sixteen data centers located in different geographic regions. Every data center consisted of five physical machines. Tasks arrive randomly on each data center with service times exponentially distributed. For each simulation setup, we generated twenty different task distributions. The physical machines and the virtual machines configurations of all scenarios are stated in Table I.

	Parameter	**Values**
Scenario 1	Length of tasks	10^4 - 8×10^4 MI
	Total number of tasks	200
	Number of VMs	100
Scenario 2	Length of tasks	10^4 - 8×10^4 MI
	Total number of tasks	1200
	Number of VMs	100
Scenario 3	Length of tasks	8×10^4 MI
	Total number of tasks	200
	Number of VMs	100
Scenario 4	Length of tasks	8×10^4 MI
	Total number of tasks	1200
	Number of VMs	100

TABLE I: Simulation parameters.

The metrics which characterize the performance of the algorithms are the *Overhead*, the *Completion Rate*, and the *Average CPU Utilization*. The fault tolerance *overhead* is computed as follows:

$$Overhead_{RPMFT} = \frac{L_{RPMFT}}{L_{FFLB}} \tag{5}$$

where L_{RPMFT} is the latency achieved by the fault tolerant algorithm, and L_{FFLB} denotes the latency achieved by the fault-free version of the algorithm.

The *Completion Rate* is the number of completed tasks divided by the total number of tasks received by the system, while the *Average CPU Utilization* is the third metric used to assess the algorithm's performances which is defined as:

$$ACU = \frac{\sum_{T_k \in RT} CU_{T_k}}{|RT|} \tag{6}$$

where RT is the set of received tasks and CU_{T_k} is the amount of CPU usage when processing a task T_k.

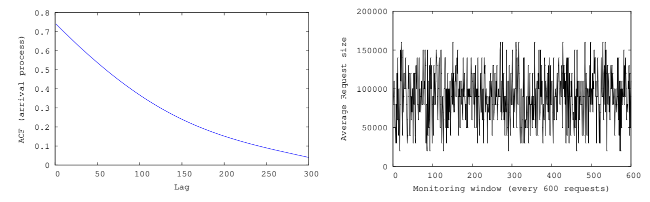
The time-to-failure of the machines is represented with a mathematical model that describes the probability of failures occurring over time which is called Weibull distribution. It is also known as the *Probability Density Function* (PDF) which is a general purpose reliability distribution used to model material strength, time-to-failure of electronic and mechanical components, equipment or systems. The Weibull failure rate function, $\lambda(x)$, is given by:

$$\lambda(t) = \frac{\beta}{\eta}\left(\frac{t-\gamma}{\eta}\right)^{\beta-1}, t \geq \gamma \geq 0, \beta > 0, \eta > 0 \tag{7}$$

where β and η are known as the shape and the scale parameters respectively, and γ is known as the location parameter.

For our experiments, we assume that the failures increase over time. For this reason, the shape parameter β is fixed to 3. The location parameter γ is set to 0 in order to trigger failures without delay and the scale η is set to 100.

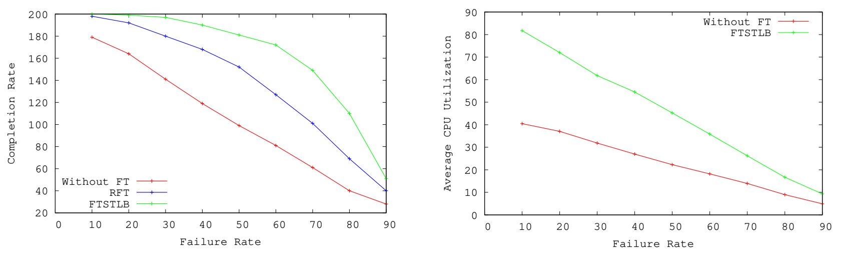
The RPMFT algorithm is simulated to explore its performance to achieve more completion rate and less overhead and average CPU utilization. The proposed algorithm was compared to the replication fault tolerance (RFT) approach in terms of completion rate.



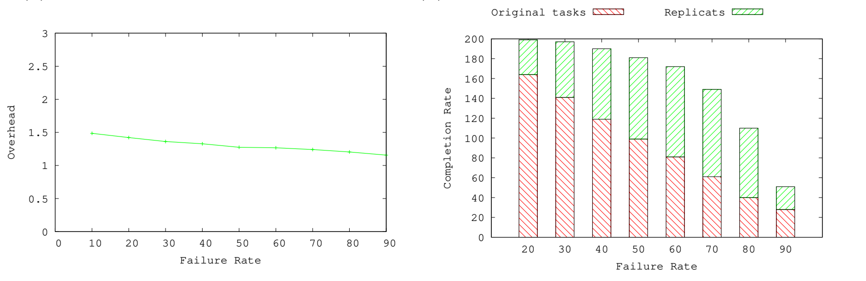
(a) ACF for tasks arrival process (b) Average request size.

FIG. 3: ACF for arrival process used in the simulation (a), and (b) Average request size for every 600 requests for tasks with random length.

Throughout this paper, we use the Autocorrelation Function (ACF) as a metric of the dependence structure of request arrivals. The ACF's decay rate determines if the tasks arrival process exhibits weak or strong correlation. Figure 3.(a) presents the ACF of tasks arrival.

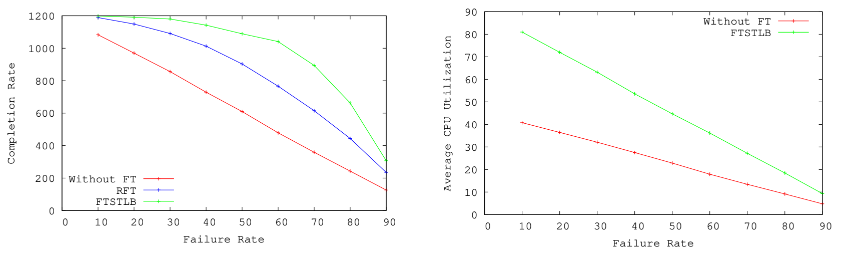


(a) Completion rate versus failure rate. (b) Average CPU utilization versus failure rate.

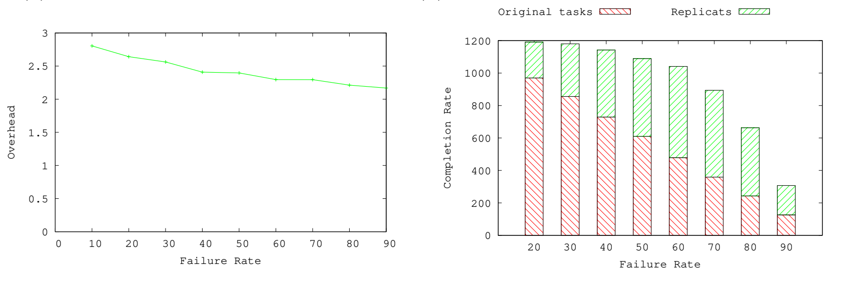


(c) Overhead versus failure rate. (d) Completed tasks origin versus failure rate.

FIG. 4: Completion rate (a), Average CPU utilization (b), Overhead (c) vs. failure rate, and (d) Completed tasks origin vs. failure rate for a set of 200 tasks with random length.



(a) Completion rate versus failure rate. (b) Average CPU utilization versus failure rate.



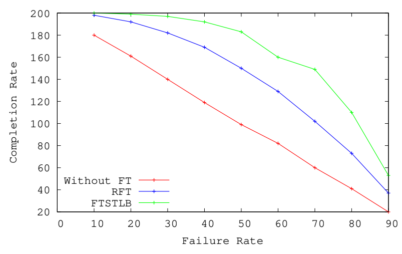
(c) Overhead versus failure rate. (d) Completed tasks origin versus failure rate.

FIG. 5: Completion rate (a), Average CPU utilization (b), Overhead (c) vs. failure rate, and (d) Completed tasks origin vs. failure rate for a set of 1200 tasks with random length.

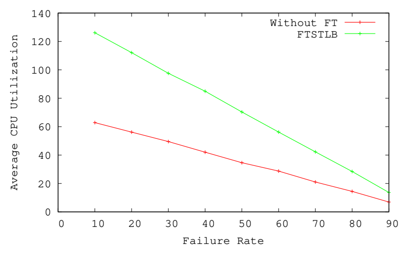
Figure 3.(b) shows the requests size window every 600 requests for Scenario 1 and Scenario 2. While Figure 7.(b) shows the requests size window for Scenario 3 and Scenario 4.

The results presented in Figure 4 were obtained using 100 VMs and a set of 200 tasks with random sizes (10^4 MI - 8×10^4 MI). Figures 4(a) shows a significant improvement in completion rate. However, the performance of all algorithms converge to the same completion rate when the failure rate is close to 100%. On average, there are 18% improvement in terms of completion rate compared with the RFT algorithm. The Average CPU utilization of RPMFT algorithm is higher as shown in Figure 4(b) which is not surprising since we need more resources for task replications.

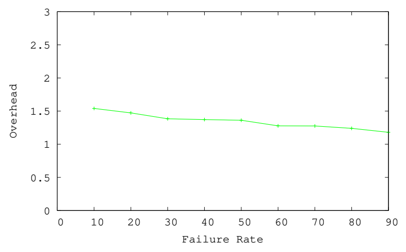
Figure 4(c) represents the fault tolerance overhead



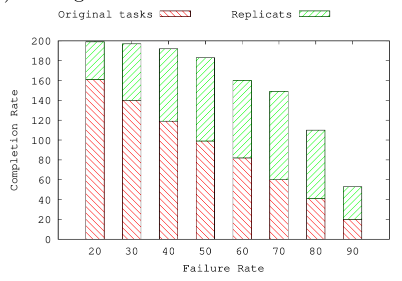
(a) Completion rate versus failure rate.



(b) Average CPU utilization versus failure rate.

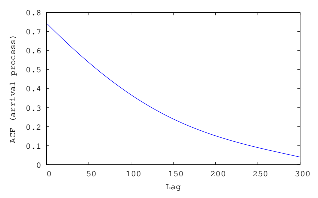


(c) Overhead versus failure rate.

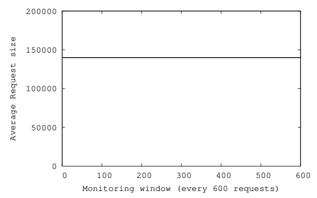


(d) Completed tasks origin versus failure rate.

FIG. 6: Completion rate (a), Average CPU utilization (b), Overhead (c) vs. failure rate, and (d) Completed tasks origin vs. failure rate for a set of 200 tasks with same length.



(a) ACF for arrival process



(b) Average request size.

FIG. 7: ACF for arrival process used in the simulation (a), and (b) Average request size for every 600 requests for tasks with the same length.

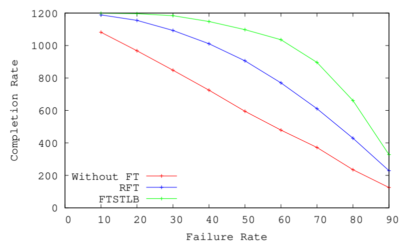
when failures occur. We readily observe from this figure that the overhead decreases slightly together with the number of failures. This is due to the fact that the fault tolerance overhead is already absorbed by the increase of failure rate, which leads to a decrease in the number of completed tasks and thus to a decrease of the induced overhead. Figure 4(d) reveals that the RPMFT algorithm reported significant number of completed tasks using replicas, especially for failure rates equal to 60%, 70% and 80%.

In the second scenario plotted in Figure 5, as the failure rate increases, the completion rate decreases notably for the RFT approach that uses replication as the unique criterion for fault tolerance. The performance of our proposal is significantly better. For instance, Figure 5(a) shows that the completion rate has been improved by 17%, and Figure 5(d) reveals also good performances in terms of the number of completed tasks. This can be explained by the use of preemptive migration for fault tolerance only if the reliability of VMs is near a prescribed level of confidence index.

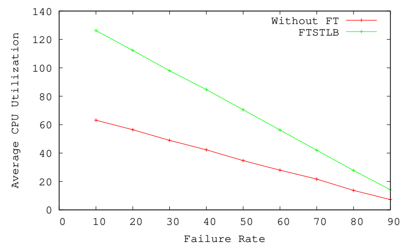
As outlined in Figure 6, the results obtained using a set of 200 tasks with the same sizes (8×10^4 MI) did not significantly change compared to the ones of scenario 1. Figures 6(a) depicts a significant improvement in completion rate. On average, there are 17% improvement in terms of completion rate compared with the replication algorithm. The Average CPU utilization of RPMFT algorithm is 100% bigger as shown in Figure 6(b). Figure 6(c) represents the fault tolerance overhead when failures occur. It is not surprising that the overhead decreases slightly together with the number of failures because it is already absorbed by the increase of failure rate, which leads to a decrease in the number of tasks completed and therefore to a reduction in the induced overhead. Finally, Figure 6(d) reveals that the RPMFT algorithm highlights significant number of completed tasks using replicas, especially for failure rates equal to 60%, 70% and 80%.

In the last scenario reported in Figure 8, it can be seen that as the failure rate increases, the completion rate decreases mainly for the RFT approach that uses replication as the unique criterion for fault tolerance. This can be attributed to the fact that RPMFT algorithm uses preemptive migration for fault tolerance only if the reliability of VMs is near a prescribed level of confidence index. Other observations indicate that the completion rate has been improved by 17% as shown in Figure 8(a) and significant number of completed tasks are task-replica as highlighted in Figure 8(d).

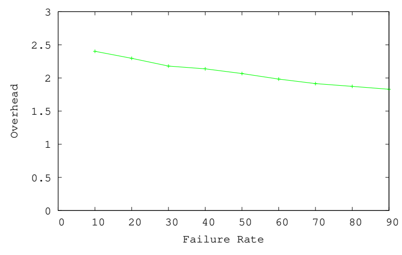
Additional tests were performed to analyse how the RPMFT algorithm deals with replication when the number of failures goes up. The results are plotted in Fig-



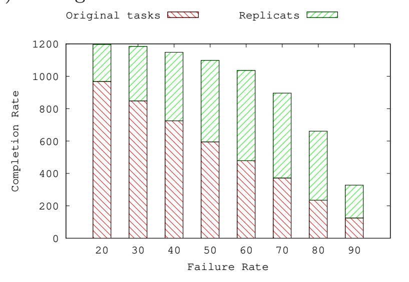
(a) Completion rate versus failure rate.



(b) Average CPU utilization versus failure rate.



(c) Overhead versus failure rate.



(d) Completed tasks origin versus failure rate.

FIG. 8: Completion rate (a), Average CPU utilization (b), Overhead (c) vs. failure rate, and (d) Completed tasks origin vs. failure rate for a set of 1200 tasks with same length.

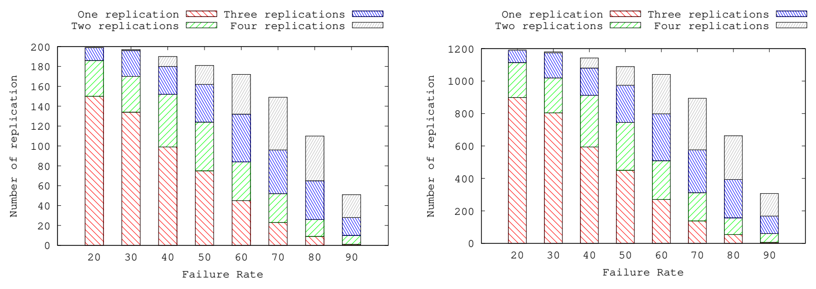


FIG. 9: Number of replication for completed tasks vs. failure rate for a set of 200 (a), and 1200 (b) tasks with random length.

ure 9. We can observe from this figure that the number of task-replicas for the same task increases together with the failure rate. For instance, for a failure rate of 20%, 81% of completed tasks were replicated one time and the remaining 19% were replicated two times. However, when the failure rate is equal to 90%, we can notice that 2%, 18%, 35%, and 45% of the completed tasks were replicated 1, 2, 3, and 4 times, respectively. Indeed, if we have a low failure rate, the reliability factor for the majority of VMs will be greater than the confidence index, which limits the number of task-replicas for the same task. On the other hand, if the failure rate increases, the number of replicas of a task also increases because the reliability factor will be lower than the confidence index for more VMs. This feature allows the RPMFT algorithm to react dynamically to failures that occur in the system, resulting in high completion rate while minimizing resources utilization.

VI. CONCLUSION

In this article, we have proposed a new fault tolerant algorithm for load balancing in cloud computing environments. Our proposal is made upon a combination of proactive and reactive fault tolerance techniques in an adaptive way. It aims to minimize the number of replications while maintaining system reliability. To this end, a reliability factor is used to decide when the migration of task-replicas should be done. Based on CloudSim simulator, it was shown that, when dealing with fault tolerance and load balancing, our algorithm exhibits better performances than its direct competitor RFT in all the tested scenarios. In future works, it would be useful to evaluate the behavior of the RPMFT algorithm in a real world cloud computing environment. Another interesting research direction is to add an artificial intelligence technique to improve the decision made when predicting failures and choosing the suitable nodes for task replication.

[1] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott, Proactive fault tolerance using preemptive migration, in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing* (IEEE, 2009) pp. 252–257.

[2] M. Pokharel and J. S. Park, Increasing system fault tolerance with software rejuvenation in e-government systems, IJCSNC, International Journal of Computer Science & Network Security **10**, 160 (2010).

[3] E. Dijkstra, ªself-stabilizing systems in spite of distributed control, º comm (1974).

[4] F. Cappello, Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities, International Journal of High Performance Computing Applications **23**, 212 (2009).

[5] K. K. Agarwal and H. Kotakula, Replication based fault tolerance approach for cloud, in *International Conference on Distributed Computing and Internet Technology* (Springer, 2022) pp. 163–169.

[6] J. Chen, X. Liang, and Z. Chen, Online algorithm-based fault tolerance for cholesky decomposition on heterogeneous systems with gpus, in *Parallel and Distributed Processing Symposium, 2016 IEEE International* (IEEE, 2016) pp. 993–1002.

[7] S. Ibrahim, B. Boulifa, S. Elloumi, A. Jaoua, M. Saleh, L. Van Den Broeke, and I. Abu-Reesh, Natural fault tolerance in the context of a goal oriented software design, in *Software Engineering and Service Science (ICSESS), 2013 4th IEEE International Conference on* (IEEE, 2013) pp. 279–282.

[8] E. Sindrilaru, A. Costan, and V. Cristea, Fault tolerance and recovery in grid workflow management systems, in *2010 international conference on complex, intelligent and software intensive systems* (IEEE, 2010) pp. 475–480.

[9] M. Gamell, K. Teranishi, M. A. Heroux, J. Mayo, H. Kolla, J. Chen, and M. Parashar, Local recovery and failure masking for stencil-based applications at extreme scales, in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (IEEE, 2015) pp. 1–12.

[10] H. Mansouri, N. Badache, M. Aliouat, and A.-S. K. Pathan, Checkpointing distributed application running on mobile ad hoc networks, International Journal of High Performance Computing and Networking **11**, 95 (2018).

[11] M. R. Malekpour, A byzantine-fault tolerant self-stabilizing protocol for distributed clock synchronization systems, in *Symposium on Self-Stabilizing Systems* (Springer, 2006) pp. 411–427.

[12] K. Plankensteiner, R. Prodan, and T. Fahringer, A new fault tolerance heuristic for scientific workflows in highly distributed environments based on resubmission impact, in *2009 Fifth IEEE International Conference on e-Science* (IEEE, 2009) pp. 313–320.

[13] A. Lakhan and X. Li, Transient fault aware application partitioning computational offloading algorithm in microservices based mobile cloudlet networks, Computing , 1 (2019).

[14] S. Chakravorty, C. L. Mendes, and L. V. Kalé, Proactive fault tolerance in mpi applications via task migration, in *International Conference on High-Performance Computing* (Springer, 2006) pp. 485–496.

[15] T. Pitoura, N. Ntarmos, and P. Triantafillou, Saturn: range queries, load balancing and fault tolerance in dht data systems, IEEE Transactions on Knowledge and Data Engineering **24**, 1313 (2010).

[16] S.-S. Wang, K.-Q. Yan, and S.-C. Wang, Achieving efficient agreement within a dual-failure cloud-computing environment, Expert Systems with Applications **38**, 906 (2011).

[17] H.-T. Chang, Y.-M. Chang, and S.-Y. Hsiao, Scalable network file systems with load balancing and fault tolerance for web services, Journal of Systems and Software **93**, 102 (2014).

[18] J. Fang, P. Chao, R. Zhang, and X. Zhou, Integrating workload balancing and fault tolerance in distributed stream processing system, World Wide Web **22**, 2471 (2019).

[19] T. Tamilvizhi and B. Parvathavarthini, A novel method for adaptive fault tolerance during load balancing in cloud computing, Cluster Computing **22**, 10425 (2019).

[20] A. R. Setlur, S. J. Nirmala, H. S. Singh, and S. Khoriya, An efficient fault tolerant workflow scheduling approach using replication heuristics and checkpointing in the cloud, Journal of Parallel and Distributed Computing **136**, 14 (2020).

[21] M. Chatterjee, A. Mitra, S. K. Setua, and S. Roy, Gossip-based fault-tolerant load balancing algorithm with low communication overhead, Computers & Electrical Engineering **81**, 106517 (2020).

[22] B. Ray, A. Saha, S. Khatua, and S. Roy, Proactive fault-tolerance technique to enhance reliability of cloud service in cloud federation environment, IEEE Transactions on Cloud Computing , 1 (2020).

[23] J. Nakamura, Y. Kim, Y. Katayama, and T. Masuzawa, A cooperative partial snapshot algorithm for checkpoint-rollback recovery of large-scale and dynamic distributed systems and experimental evaluations, Concurrency and Computation: Practice and Experience **33**, e5647 (2021).

[24] Z. Li, V. Chang, H. Hu, H. Hu, C. Li, and J. Ge, Real-time and dynamic fault-tolerant scheduling for scientific workflows in clouds, Information Sciences **568**, 13 (2021).

[25] A. Rezaeipanah, M. Mojarad, and A. Fakhari, Providing a new approach to increase fault tolerance in cloud computing using fuzzy logic, International Journal of Computers and Applications **44**, 139 (2022).

[26] M. Rahman, M. A. Rouf, *et al.*, Aggressive fault tolerance in cloud computing using smart decision agent, in *Proceedings of the International Conference on Big Data, IoT, and Machine Learning* (Springer, 2022) pp. 329–344.

[27] A. Semmoud, M. Hakem, B. Benmammar, and J.-C. Charr, Load balancing in cloud computing environments based on adaptive starvation threshold, Concurrency and Computation: Practice and Experience **32**, e5652 (2020).

[28] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. 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 **41**, 23 (2011).