Cloud-Niagara: A High Availability and Low Overhead Fault Tolerance Middleware for the Cloud

Asif Imran*, Alim Ul Gias, Rayhanur Rahman, Amit Seal, Tajkia Rahman, Farhan Ishraque, and Kazi Sakib
Email: asif.imran.anik@gmail.com, alimulgias@gmail.com,
bit0101, bit0122, bit0107, bit0201@iit.du.ac.bd, sakib@univdhaka.edu
Institute of Information Technology, University of Dhaka

***Abstract*—Fault tolerance is the ability to a system to continue its functionality despite the presence of faults in the architecture. For a dynamic system such as the cloud, fault tolerance is required to ensure business continuity. This paper proposes a high availability middleware that ensures fault tolerance for cloud based applications. Effective Descriptive Set Theory is used to determine the model of fault detection for real life applications running on the open source cloud. A deterministic algorithm of the middleware is provided that achieves automatic allocation of backup nodes to the system based on the faults. After detection of faults, the middleware directs the system to add new nodes as replicas of the failed nodes, ensuring continuity of the cloud applications. Next, a case study including seven real life applications such as PostGreSQL Database, etc are described and fault tolerance is ensured through the proposed middleware. Empirical performance analysis of the algorithm is carried out and results are compared to traditional systems.Results show that in the presence of faults induced during experimentation, the middleware can be effectively used to introduce replica and ensure fault tolerance of bottleneck resources for executing 700 to 1000 processes per unit time.**

I. Introduction

Fault tolerance is the ability of a system to perform its function properly in case of faults and excessive system loads. Cloud computing is the dynamic provisioning of virtual machine instances from a shared resource pool of physical machines. It provides dynamically distributed entities for rapid scaling and dynamic provisioning to increase performance. For this reason, fault tolerance needs to be ensured for cloud services to prevent lower level errors from propagating into system failure. Ensuring fault tolerance for a largely dynamic system such as the cloud presents significant research challenges, since a large number of processes in different states are involved. Attributes of the algorithm such as watchdogging, checkpointing and logging critical processes have not been considered for the cloud. Modest studies have been conducted to ensure fault tolerance in traditional systems [1], [2]. More specifically, the research issues that need to be addressed are provided in the following.

1) Mathematically analyze causes of faults in the cloud. Using the analyses, propose an algorithm for a middleware to ensure high availability fault tolerance of the cloud platform through rapid checkpointing, watchdogging and logging.
2) Empirical investigation to analyse the performance of the proposed algorithm of the middleware in a real life case scenario.

A modest number of reports and papers have emanated from the study to date, including a general overview [1], papers concerned with protection of distributed systems from failures were highlighted in [2], [3]. Preliminary account on fault detection and recovery through replication is presented in [4]. However, providing an algorithm for a middleware to ensure high availability fault tolerance have been considered to a limited extent.

Based on the research issues stated above, this paper proposes an algorithm of a middleware that ensures fault tolerance of cloud. The algorithm of the middleware is deterministic and automatically checkpoints and logs system faults. Upon detection of faults, the algorithm directs the system to provision replicas to faulty machines. The proposed middleware is called Cloud-Niagara because it is capable of ensuring fault tolerance at excessive system loads and existing faults through incorporating multiple primary controllers and using dynamic group architecture. The benchmark for industry level Central Processing Unit (CPU) and Random Access Memory (RAM) is considered to be 70% [5]. Effective Descriptive Set Theory is used to model the occurrence of faults in the cloud [6].

In the case study, seven real life applications namely Online Survey application (OSa), Source Code Repository (Rep) applications, Open Source Project Manage-

ment application called HTProjects (Proj), PostgreSQL Database (DB) Management Applications, File Transfer Protocol (FTP) services, Nova Dashboard (ND) for OpenStack Compute Cloud and Domain Name Services (DNS) were executed in the cloud. Next, Cloud-Niagara was applied in the distributed controllers and used to monitor the resource consumption of the applications. Additional instances were placed to act as replicas to achieve fault tolerance. The results obtained from the case study are used to show how the middleware performs in achieving fault tolerance in the cloud. The results reflect system load exceeding 70% when no replica or fault tolerance mechanism is in place for executing 700 to 1000 processes concurrently per unit time.

II. Related Work

Preventing failures in systems through detection of peer failures and obtaining lost information in large scale Distributed Hash Tables is proposed in [1]. Horizontal Active Replication scheme is presented that ensures viewing key processes to detect failures. The mechanism ensures an effective range of query processing, even when probability of failure is significantly high. However, load balancing and failure detection is carried out by a master load server that may lead to low availability.

Linear Finite State Machines (LFSMs) have been made fault tolerant in discrete time dynamic systems, such as simulations [2]. Redundant implementations are used to construct fault tolerant systems, that involve separate machines to preserve state, evolution and properties of target systems. The redundant machine protect the data of the target nodes through encoding. However, the communication between the remote machines that ensure redundancy result in high latency of the architecture.

Automatic addition of new fault handlers into an existing fault tolerance scheme is proposed in [3], [4]. The assumption is based on the problem of anticipating faults of all program classes at the time of system design. The methodology involved updating new faults at the system design level through incorporating of a random fault generating module. Significant contribution of effective resource provisioning have been made in [7]. However the performance benchmarks of resource utilization were not considered.

Fault tolerance is achievable means of active redundancy as proposed in [5]. Dependability analysis needs to be carried out to analyze the degree of redundancy. Self-checkers are installed that is used to differentiate the faults. The self-checker is required to be intrinsic and fail-stop activities of the component based distributed system needs to be placed. However, each message in the methodology is required to have multiple components executed physically which results in increased traffic.

Protocols like Isis, Amoeba and Totem deliver messages similarly in the total order [8]. Those protocols are widely used to maintain consistency in replication of fault tolerant systems [9]. However, delays are introduced when atomic multicast potocols are used. Delays in messaging protocols can be prevented if the primary is responsible for deciding the ordering of the operations. However, ensuring the high availability of the primary node through a middleware is a significant challenge. The high number of message passing in synchronous communication will be very costly for the consensus algorithm.

III. Fault Standard of Cloud-Niagara

As discussed in the previous section, ensuring high availability fault tolerance requires a middleware that can monitor resource usage and send notifications without delay. The proposed architecture of Cloud-Niagara satisfies the requirement discussed here. Cloud application failures arise from incorrect program designs or coding errors but, more often than not, they arise from transient and nondeterministic errors. A mathematical model needs to be provided to analyse the type of faults.

A. Effective Descriptive Set for Fault Identification

The system S_n consists a finite set of real life application A_i where $A_i = \left(a_1, \quad a_2, \quad \cdots \quad , a_u\right)$. Each application a_i consists of a finite set of processes P_j denoting set of all processes such that the condition $\forall a \in A : P_j = \left(p_1, \quad p_2, \quad \cdots \quad , p_v\right)$ is satisfied where *u* and *v* are positive integers. Each process p_c have states that are represented by a finite set $S_t = \left(s_1, \quad s_2, \quad \cdots \quad , s_n\right)$. The states are divided into two subsets representing the correct and failed sets respectively, where the value is obtained by $S_{correct} = \left(s_{l1}, \quad s_{l2}, \quad \cdots \quad , s_{ln}\right)$ and incorrect state as $S_{failed} = \left(s\prime_{l1}, \quad s\prime_{l2}, \quad \cdots \quad , s\prime_{ln}\right)$. Hence the above cases are true if and only if

$$(S_{correct} \bigcup S_{failed} \in S_t) \wedge (S_{correct} \bigcap S_{failed} \in \emptyset)$$

are true. The combined states of the various processes in the real life applications running on the cloud platform is satisfied by the following equation 1.

$$\begin{aligned} B_i = (\forall i \in 0 < i \leq n : b_i \subset S_{correct}) \wedge \\ (\forall j \in 0 < j \leq n : b_j \subset S_{failed}) \end{aligned} \tag{1}$$

Valid write counts is projected as $(s_l, s\prime_l) : S_t \in \delta_p \wedge s_l, s\prime_l \in S_t$. Here δ_p is the probability of a valid state. Write operations are valid if and only if a process p_j executes operation w_j on the file that is allowed access.

Hence, if $w_j\prime$ is a set of write operations that result in process failure, we get,

$$w_j\prime = (s_l, s_l\prime) : (\exists p_j : x \notin w[j] \wedge x = S_t) \quad (2)$$

Read operations can be carried out in multiple files. Hence the files that are in specific group denoted by G(R_k) is described here.

$$\begin{aligned} G(R_k)(s_l, s_l\prime) &= ((s_l\prime) : \forall y : y \in r_y : v(s_l) \\ \neq v(s_l\prime) \vee v(s_l\prime) &= v(s_l) \vee \forall y : y \notin r_y : v(s_l) \\ &= v(s_l\prime) \wedge v(s_l\prime) \neq v(s_l)) \end{aligned} \quad (3)$$

An application a_i $(0 < i \leq u)$ consists of a process p_j $(0 < j \leq v)$ that carries out read and write operations on the data stored in the cloud. To ensure low latency fault detection by proposed architecture, faults must be detected with minimal delay in which those occur [3]. Read operations are represented by the set R_x and write operations are represented by the set W_y. After read or write operation by p_j, the state is saved in a state variable s_l and updated in S_t. If the state is faulty the $S_t : s_l = s\prime_l$ else $S_t : s_l$. Hence, the conditions need to satisfied if s_l is to be correct and not failed,

$$\begin{aligned} &\exists p \in P_j : G(s_l) \subseteq S_t \vee G(s_l) \subseteq S\prime_t) \\ \implies &G(s_l) \subseteq R_x \wedge G(s_l) \subseteq R\prime_x = \emptyset \vee \\ &G(s_l) \subseteq W_y \wedge G(s_l) \subseteq W\prime_y = \emptyset; \end{aligned} \quad (4)$$

IV. Proposed Algorithm for Cloud-Niagara

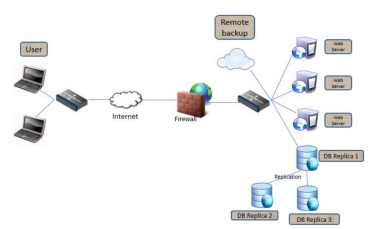
High availability fault tolerance is ensured by Cloud-Niagara through resource utilization monitoring and notifying the primary controller. The Algorithm 1 presents the algorithm of Cloud-Niagara and identifies the input and output of the distributed architecture. The algorithm notifies the system to add new nodes once the utilization of resources exceeds the predefined threshold. N_{node} is a queue that holds the node id of each node that is allocated to work with the cloud system. For each entry in N_{node}, the ratio of $R_{current}$ that is currently used resource of a node is calculated. The total resource available of that node is denoted by R_{actual} that is calculated simultaneously. Here M is the maximum number of resources available and $N_{temporary}$ are the additional resources added for a specific time period t.

The obtained value of R_{actual} is checked against upper threshold value T_{upper}. If the ratio is more than or equal to the threshold value, the node identity is added to another queue N_{exceed}. Else it is added to another queue N_{low}. Finally these queues are sorted in ascending order of node identities.The following section highlights a real life case study used for testing the proposed algorithm of Cloud-Niagara. Out of the total number of processes, let H be the number of processes that are faulty at time

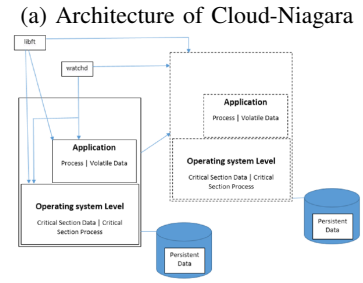
Algorithm 1: Algorithm of Cloud-Niagara

Input: Resource Usage data from the slave nodes $\langle a_1, a_2, \ldots, a_n \rangle$
Output: Graphical of output of resource usage information and warning about faulty nodes

```
location ← 0
i ← 2
while n ← N_nodes do
    if R_current/R_actual ≥ T_upper then N_exceed ← N_exceed + 1;
    else R_current/R_actual < T_lower N_low ← N_low + 1;
Record and sort all indexes J in ascending order of R_current/R_actual;
if N_exceed == N_node then
    Generate Warning for admin for adding more physical resource;
    N_node = N_node − 1;
    Add new node to Cloud with back up id I;
    J[i] = J[i] + 1;

if N_now ≥ 2 then
    Set m equal first index in J;

if M == 0 then
    N_node = N_node − N_m;
    Res[i] = Res[i] − Res[j]
    N_node = N_node − 1;
    N − low = N_low − 1 and J[i] = J[i-m];

if N_low ≥ 2 then
    n = J[1];
    N_node = N_node − N_temporary;
    N_node = N_node − 1;
    BackID ← 0

while x ← N_nodes do
    N_x = BackID;
return Graphical output of resource usage at different nodes
```



(a) Architecture of Cloud-Niagara



(b) Watchdog, Checkpoint and Log mechanisms of Cloud-Niagara

Fig. 1: Fault Tolerant Mechanism of the proposed solution

T that eventually leads to failure of the cloud service, hence $\lceil \frac{N}{2} \rceil \leq H \leq \lceil N \rceil$ is satisfied where N is the total number of nodes. Initially, no system has failed N nodes function correctly. Let a failed state satisfying the condition H is denoted by F.

$$\Delta_s = (N - q) * 0.30 \tag{5}$$

$$\Delta_s = (N - q) * 0.30 + \Delta_{primary} \tag{6}$$

The system can be in complete functional state when N nodes are active. Next, we have a condition where q nodes satisfy H. We see that (N-q) nodes are still active where $(0 \leq q \leq F_{max})$, where F_{max} are the maximum number of failed nodes. W define an acceptable state of machine as $(0 \leq q \leq 0.30N)$. Therefore, $F_{max} \leq 0.30N$ condition is required to be satisfied for the system to function despite system faults. The coverage factor c is defined as 0.30 since the benchmark considered for fault tolerance is 70%. The Δ_s is defined in equation 4 and Δ_s when main node fails is denoted by equation 5. The failure detection is captured by Cloud-Niagara using the mathematical model described here. The functionality of the model ensures high availability and low latency of the fault tolerance scheme. The time taken to recover from the failure is required to be compared to identify the performance of Cloud-Niagara compared to traditional mechanisms.

To compare performance of Cloud-Niagara with existing architectures, Mean Time To Recover (MTTR) from the failed state back to functionality is compared.

$$\begin{aligned} MTTR_q &= \frac{\sum_{q=1}^{n}(q)(dT)^2}{q} \\ &= \frac{\sum_{q=1}^{n}(q)(T_{recover} - T_{failure})^2}{q} \end{aligned} \tag{7}$$

The time for failure is the product of coverage factor c and number of failed nodes H. Hence

$$T = c \times H : \big(h_1, \quad h_2, \quad \cdots \quad , h_n\big) \leq 0.30N \tag{8}$$

Cloud-Niagara uses active replication technique to make the system fault tolerant. This technique gives all the replicas the same role unlike the primary-backup technique [10]. A client process pi invokes operation op_{arg} to one replicated server x. All the replicas of x will receive the invocation op_{arg}. Next, all of them will send the response to the p_i process. The process can function if replicas do not behave maliciously.

V. Case Study of Real Life Applications for Fault Tolerance

The performance of Cloud-Niagara is evaluated through a case study on real life Openstack cloud. The middle-ware is installed in the cloud controller.

```
Cloud-Niagara: info: [2874] dispatching hogs: 0 cpu, 0 io, 7 vm, 0 hdd
Cloud-Niagara: FAIL: [2880] (495) hogvm malloc failed: Cannot allocate memory
Cloud-Niagara: FAIL: [2874] (395) <-- worker 2880 returned error 1
Cloud-Niagara: WARN: [2874] (397) now reaping child worker processes
Cloud-Niagara: FAIL: [2874] (452) failed run completed in 1s
```

Fig. 2: Resource Overloading fault warnings obtained from Cloud-Niagara

A. Scenario

The cloud infrastructure has 10 physical nodes and a controller. The middleware monitors resource usage for all the physical nodes in the cloud. The processes include running seven services discussed previously that are commonly used in real life [11].

The experiments were carried out in the cloud infrastructure of a real life cloud service provider using OpenStack Essex Cloud on Ubuntu 12.04 Long Term Service (LTS) server. The experiments in the case study included running the applications in virtual machine instances of the cloud [12]. Each instance is allocated 2 GigaBytes of Random Access Memory (RAM) and 2 Central Processing Unit (CPU) cores. The RAM and CPU are increased or decreased based on the information obtained from the proposed algorithm.

B. Bottleneck Resources

Forking processes are executed on the virtual machine instances of the cloud to load the CPU and RAM. As part of the assumptions, we also consider that the cloud infrastructure can be scaled both vertically and horizontally. The Cloud-Niagara middleware operates on the cloud controller that acts as the primary.

Cloud-Niagara monitors the system utilization and provides the resource consumption in graphical format. Once the memory allocation or CPU consumption by each core exceeds 70%, failing and warning messages are exhibited by Cloud-Niagara as shown in the Figure 2. Bandwidth is not a bottleneck when it comes to the scenario of the case study [13].

C. Methodology to Add and Remove Fault Tolerant Resources

Once a backup is there in place it needs to be added into the system. Cloud-Niagara manages addition of new backups in the following format.

1) Introducing new backup resource using Cloud-Niagara: The backup machine sends a message using multicast protocol to the internally connected group. The contents of the message involves a Backup-ID. Next the controller that acts as the primary will make a priority list in a queue and send the list over the internal network.

The Backup-ID is used to notify the accepted backup machine that responds with an acknowledgement. Then the new backup is integrated that begins to update itself from the latest check-pointed state. The checkpoint and watchdog daemons play critical roles in this regard [14]. Once the new membership is acknowledged, all processes are sent a notification message about the new core or memory unit and mapped to a specific node.

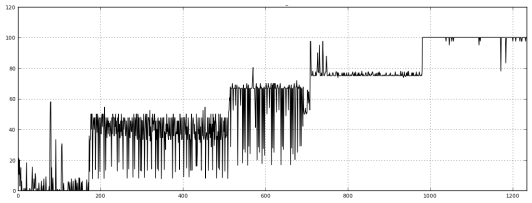
2) Removing existing backup using Cloud-Niagara: As a node becomes faulty it must be replaced. The backup removal procedure of Cloud-Niagara is replaced using broadcast messages. The primary node issues a message to remove a specific backup with a unique Backup-ID [15]. This message is broadcasted in the internal cloud network [16]. The nodes can be removed as a high priority machine requires more resources.

D. Analysis of Obtained Results

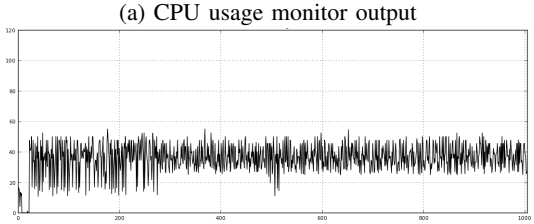
As replicas are activated, it is seen that the resources are capable of taking in increased CPU and Memory loads as depicted in Figure 3a and Figure 3b. The accumulated results are shown in Table 1 and graphs denote the number of processes triggered per unit time on the horizontal axis. The vertical axis denotes the percentage of resource utilization for the specified number of CPU cores and RAM. The figures show that the systems gradual memory and CPU consumption can be kept below the benchmark level by adding more replicas. The ring architecture of the fault tolerant nodes contributes to fault tolerance of the cloud service [11].

Initially the CPU load is under the danger level. Figure 3a shows the gradual growth of CPU utilization as the backup processors are added in accordance to the demand of the primary CPU. When no replica is used, an increased load on the CPU will result in the single Core being continuously used to full potential [17]. Increase of load at 100 units results in a sudden increase of resource utilization in Figure 3a. When the load per unit time is increased to 180 units, the CPU usage is normal. The situation remains the same till 520 processes executed per unit time. During this period, the CPU utilization of the instances remains at around the 50% scale because of provisioning by Cloud-Niagara. From 520 processes to 700 processes, the CPU load is seen to increase from 50% to 70% benchmark with a sudden increase to 80% for 580 processes per unit time.

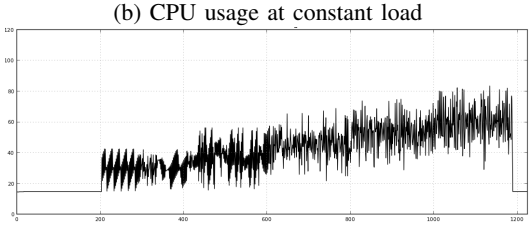
The graph in Figure 3b calculate average CPU utilization for FTP and DNS are shown in Figure 3b. For the two services, for 20 processes executed per unit time, the CPU utilization declined momentarily. This is caused by the sudden decrease in DNS and FTP processes as no user required to establish an FTP or execute DNS for that



(a) CPU usage monitor output



(b) CPU usage at constant load



(c) Graph of memory utilization monitored through Cloud-Niagara

Fig. 3: Results obtained for Cloud-Niagara

time period. From 20 processes per unit time onwards the resource usage increased to 50% and remained at the desired scale of 30% to 50% that is below the prespecified threat level of 70%. The cloud infrastructure sustains the load for upto 1000 processes executed per unit time.

Constant increase in RAM usage is identified by Cloud-Niagara and represented graphically as shown in . The graph in Figure 3c highlights the memory utilization as increased load is applied. The system increases load significantly from 0 to 1200 processes per unit time. Initially, the utilization of RAM is negligible for the 200 processes. This is because the first 200 processes are CPU intensive and consumed memory of upto 18% only. At the load of 1000 forking processes, the system load reaches 80% to 90%, well above the threshold level of 70%. This shows that the current physical memory will remain in the danger level above 70% if the processes executed per unit time ranges from 900 to 1200. Such load will cause the system to become more prone to failure.

The important information derived from Cloud-Niagara can be used to decide when to add new resources to the existing cloud infrastructure. Hence the cloud architecture can be made fault tolerant using Cloud-

Service	Cloud VM	Replica Available	Evaluation Time	Throughput	Num.of Fault	Allocate Replica
OSa	1	1	500X9	50	17	No
Rep	2	1	1000X5	67	19	No
Proj	1	1	700X7	96	12	Yes
DB	4	2	2000X6	87	20	Yes
FTP	1	1	600X7	94	12	Yes
ND	3	2	600X4	48	12	No
DNS	2	1	400X8	76	27	No

TABLE I: Obtained results of fault detection by Cloud-Niagara

Niagara's algorithm.

VI. Conclusion and Future Work

This paper aims to provide a high availability and low latency fault tolerance middleware for the cloud. The paper presents a mathematical model based on Effective Descriptive Set Theory for fault detection. In addition, the replication standards and empirical models for performance analysis of Cloud-Niagara is provided. Next, a deterministic algorithm based on watchdogging, checkpointing and journaling has been discussed for the middleware. Dynamic group communication of the algorithm in terms of read and write operations ensure effective fault tolerance.

Real life case study is shown that reflects the performance of Cloud-Niagara middle-ware in replication of CPU and Memory units. Overall utilization of the cloud service is kept below the pre-specified benchmark and resources are made more fault tolerant through a component based architecture. The results reflect that resource utilization can be effectively monitored using Cloud-Niagara and replica can be added or removed according to the need of the primary. The empirical cost estimation of Cloud-Niagara in terms of scalability and performance is a subject of future research interest.

VII. Acknowledgment

This research has been supported by the University Grants Commission, Bangladesh under the Dhaka University Teachers Research Grant No-Regi/Admn-3/2012-2013/13190.

References

[1] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira, "On fault representativeness of software fault injection," *Software Engineering, IEEE Transactions on*, vol. 24, no. 7, pp. 1313–1327, 2013.

[2] T. Pitoura, N. Ntarmos, and P. Triantafillou, "Saturn: range queries, load balancing and fault tolerance in dht data systems," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 24, no. 7, pp. 1213–1229, 2012.

[3] R.-T. Wang, "A dependent model for fault tolerant software systems during debugging," *Reliability, IEEE Transactions on*, vol. 61, no. 2, pp. 504–515, 2012.

[4] W. Xu and T. Zhang, "A time-aware fault tolerance scheme to improve reliability of multilevel phase-change memory in the presence of significant resistance drift," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 19, no. 8, pp. 1357–1367, 2011.

[5] B. Littlewood and J. Rushby, "Reasoning about the reliability of diverse two-channel systems in which one channel is," *Software Engineering, IEEE Transactions on*, vol. 24, no. 7, pp. 1313–1327, 2011.

[6] H. Becker and A. S. Kechris, *The descriptive set theory of Polish group actions*. Cambridge University Press, 1996, vol. 232.

[7] B. Javadi, J. Abawajy, and R. Buyya, "Failure-aware resource provisioning for hybrid cloud infrastructure," *Journal of parallel and distributed computing*, 2012.

[8] R. Jhawar, V. Piuri, and M. Santambrogio, "Fault tolerance management in cloud computing: A system-level perspective," *Systems Journal, IEEE*, vol. 7, no. 2, pp. 288–297, 2013.

[9] D. R. Avresky, S. J. Geoghegan, and Y. Varoglu, "Evaluation of software-implemented fault-tolerance (sift) approach in gracefully degradable multi-computer systems," *Reliability, IEEE Transactions on*, vol. 55, no. 3, pp. 451–457, 2006.

[10] T. Alt, H. Appelshauser, S. Bablok, B. Becker, S. Chattopadhyay, C. Cheshkov, C. Cicalo, J. Cleymans, R. Fearick, H. Helstrup *et al.*, "Benchmarks and implementation of the alice high level trigger," *Nuclear Science, IEEE Transactions on*, vol. 53, no. 3, pp. 854–858, 2006.

[11] A. Imran, A. U. Gias, and K. Sakib, "An empirical investigation of cost-resource optimization for running real-life applications in open source cloud," in *High Performance Computing and Simulation (HPCS), 2012 International Conference on*. IEEE, 2012, pp. 718–723.

[12] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, "Fault-tolerance in the borealis distributed stream processing system," *ACM Transactions on Database Systems (TODS)*, vol. 33, no. 1, p. 3, 2008.

[13] J. A. Duraes and H. S. Madeira, "Emulation of software faults: a field data study and a practical approach," *Software Engineering, IEEE Transactions on*, vol. 32, no. 11, pp. 849–867, 2006.

[14] X. Liu, Q. Wang, S. Gopalakrishnan, W. He, L. Sha, H. Ding, and K. Lee, "Ortega: An efficient and flexible online fault tolerance architecture for real-time control systems," *Industrial Informatics, IEEE Transactions on*, vol. 4, no. 4, pp. 213–224, 2008.

[15] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[16] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.

[17] J. H. Abawajy, "Fault-tolerant scheduling policy for grid computing systems," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 238.