BYZANTINEWARRIOR: A Byzantine Fault Detection Framework for Openstack Cloud Using Unstructured Log Analysis

FARHAN ISHRAQUE KHAN
BSSE 0201

A Thesis
Submitted to the Bachelor of Science in Software Engineering Program Office
of the Institute of Information Technology, University of Dhaka
in Partial Fulfillment of the
Requirements for the Degree

Bachelor of Science in Software Engineering

Institute of Information Technology
University of Dhaka
DHAKA, BANGLADESH

BYZANTINEWARRIOR: A Byzantine Fault Detection Framework for Openstack Cloud Using Unstructured Log Analysis

FARHAN ISHRAQUE KHAN

Approved:

Signature *Date*

_____ _____

Supervisor: Dr. Kazi Muheymin-Us-Sakib

_____ _____

Committee Member: Dr. Shariful Islam

_____ _____

Committee Member: Dr. Mohammad Shoyaib

To *Mozammel Haque Khan and Razina Akhter* my respective parents who have been my constant source of inspiration. They have given me the drive and discipline to tackle any task with enthusiasm and determination. Without their love and support this project would not have been made possible.

Abstract

Open source cloud computing provides free, on-demand, rapidly scaling, ubiquitous computing and data storage services that promises a significant number of prospective customer base. However, the largely distributed nature and growing demand for open source cloud makes the infrastructure hard to assure availability and performance. The cloud administrators are continuously concerned about service disruption due to failure.

Though fault tolerance in distributed systems have been studied for a long while and came to some certain solutions, there still exists the threats of arbitrary failure that can cause system down time. The main adverse side of arbitrary failure or Byzantine Failure is that it does not show up until the last moment of failure. But, this sort of failure causes continuous deterioration of performance of the system. Though this is an active area of research in the field of fault tolerance, little work is done in this context.

ByzantineWarrior is an effective Byzantine Fault detection skim for cloud that ensures the detection of faulty process in the highly distributed systems like cloud. ByzantineWarrior analyze the console log files of various cloud modules like nova-compute, scheduler and create Finite State Machine(FSM). After defining the standard FSM, ByzantineWarrior can effectively analyze the execution of process and cross match against it.

The research analyzes the possibilities of detecting Byzantine Failure in cloud. Since Openstack cloud comprise of multi-modules, the research was done for specific modules namely *nova-compute, nova-scheduler, neutron-network and glance-api.*The obtained results of the experiments were compared against standard benchmarks to identify a performance of average 90.70 percent precision for the test cases. A significantly low overhead of -0.219 was obtained through tests carried out with standard baseline values which prove the generalized capability of Byzan-

tineWarrior to detect Byzantine Fault in Cloud.

The conclusion of this research suggests the effectiveness of the ByzantineWarrior algorithm and scope of improving the scheme to industry standard fault detection system of open source cloud computing such as traditional fault detection techniques.

Acknowledgments

This thesis is about devising a scheme for Byzantine Fault detection in open source cloud that highlights exactly my research interest for the last one year. Although the thesis is one semester long, I have been doing my research on this topic for almost one year now. It has been mainly possible for the constant effort of my thesis supervisor Dr. Kazi Muheymin-Us-Sakib sir, whose continuous support has enabled me to take this work into reality. It all began in December 2012 when our respected program chair, head of intern office and now my research supervisor Dr. Sakib sir sent me to Panacea Systems Ltd. to pursue my internship. It was at Panacea Systems Ltd. where I got my first flavors of cloud computing and there was no looking back from then on. I am extremely grateful to Dr. Sakib sir for lending me this opportunity. From that moment I started to learn about the cloud using Google to good effect. I connected to many people who deal with this technology, electronically and face to face. Since it is impossible for me to thank them all, I will take this opportunity to mention the names of those without whom this study would have never been completed.

This work is a reality because of honorable Dr. Sakib sir. From the very begining, he has been guiding me in the research field, identifying my mistakes by going through every line of my write ups and my experimental results, investing significant portions of his valuable time to my development. At the same time he was always concerned about keeping my motivation high and maintaining a high spirit within the research team. Dr. Sakib sir, without your guidance I would have been lost in this vast research field a long time ago. Thank you so much for supervising, encouraging and helping me in the research and taking the burden and responsibility of reviewing my findings so that they gain acceptability.

I am also very grateful to Dr. Mahbubul Alam Joarder sir, in his role as the Director of the Institute of Information Technology, University of Dhaka, for allowing

me pursue my research in a healthy and competitive atmosphere of IIT. I will also take this opportunity to thank Dr. Md. Mohammad Shoyaib sir and Shah Mostafa Khaled sir for supporting me and helping me doing my work till the late night in IIT.

As I said earlier, I first learnt the cloud at Panacea Systems. I would like to Thank Mr. Asif Imran Anik for helping me doing my job and carrying out my research work at the same time. He has been a second supervisor for me. Also, my heartiest thanks to Mr. Redwanul Karim Ansari, Chief Executive Officer (CEO) of Panacea Systems Ltd, for appointing me as the Cloud Artisan, Cloud Computing at his company. Hence I was fortunate enough to be provided with the powerful cloud servers of Panacea where I executed the tests and obtained the results of the thesis experiment.

Finally I would like to thank my family and friends who have directly and indirectly helped me in the research. Special thanks go to my mother, who have supported me by any means at any time. My elder brothers Zadid Raiyan and Sadat Rayan has given me enormous support from my family.I would like to thank all the students of BSSE02 batch who are seeking completion of the Bachelor of Science in Software Engineering (BSSE) program through their participation in research and projects. Thanks go to the staffs of IIT, who have been a continuous source of help. The members of IIT family have really made this research fun for me and made the moments very enjoyable.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cloud is an evolving technology, the basis of which is virtualization of physical servers to form virtual machines (vm) instances [1]. Significant number of processes executed on the cloud is based on workflows, meaning it involves a series of processes which generate many intermediate data products [2]. As cloud is a form of distributed systems, it has always been prone to Byzantine faults [2]. Detection of Byzantine faults for workflow based processes is essential for cloud to ensure resilient and trustworthy monitoring.

The purpose of this study is to develop a Byzantine fault detection model for the cloud environment to track the anomaly behavior of cloud. Besides controlling emphatic number of processes and modules (for example: nova-compute, quantum network, glance etc) on the cloud, the challenge of making the inter process workflow precisely is equally important for business continuity of cloud [3]. This work will also aim to achieve this challenge by developing a user-centric Byzantine fault detection model for cloud.

Byzantine Fault detection can help detecting execution anomaly, work flow error and low performance [**?**] of cloud. Currently console logs are used to detect such types of error. But, console logs rarely help operators detect problems in large-scale data enter services. Because, they often consist of the voluminous intermixing of messages from many software components written by independent developers [4]. Though, there are cloud monitoring services like HP Openview, Microsoft SCOM and Amazon Cloudwatch, they do not provide monitoring schemes for Byzantine fault [5]. Moreover, these applications work in a centralized way making them prone to single point of failure [5]. Hence a model for extracting execution anomaly detection for cloud must be addressed [6].

Cloud administrators need to keep track of the process workflow and anomalistic

behavior to keep cloud system up and running. Hence, every unusual log reports (warning and errors) must be tracked for fault detection manually [4]. Moreover, desired performance benchmark must be obtained for production level cloud.

1.1 Addressing The Research Problem

Although extended studies have been carried out in fault tolerance [7] [8][9], the main challenges and requirements to identify Byzantine fault effectively and automatically is an active area of research. Considering the dynamic and complex nature of cloud computing, a model that would link log and audit data collected from multiple infrastructures is a challenging task [7]. These challenges should be identified to a greater detail in order to ensure that they can be mitigated for reliability of cloud computing. Challenges of automated anomaly detection can be mitigated and at the same time user involvement and increased semantic knowledge on Byzantine fault can be ensured [4]. Assurance that specific Byzantine fault will not lead to system failure is still an open research question [9]. According to current scenarios, there is no effective way to detect Byzantine Fault in the application layer of Openstack cloud. Finding a way to detect these sorts of faults in Openstack is very important to ensure availability and performance. This work will concentrate on an effective way of detecting Byzantine faults of cloud's system modules.

1.2 Byzantine Failure Overview

Byzantine Failure occurs in a system when components of a system Fail in arbitrary way(the system does not stop or crush but may process request incorrectly) corrupting their local state and/or producing incorrect or inconsistent outputs[].In

standalone systems, Byzantine failure can not propagate to large scale failure. However, In parallel systems, it often results in avalanche effect. Apart from this scenario, there is no other effective way to detecting this sort of failure.[5]

In most cases effects of Byzantine Failure is hidden. But even in the hidden mode, this failure can cause performance degradation[6]. For services like Cloud, where availability and integrity are key point of concern, detecting Byzantine Failure is of significant importance. So Byzantine Failure can be of two types: omission failure and commission failure. In *Omission Failure* the system halts and stops sending/recieving data. On the other hand, incorrect process request or corrupted request is generated in *commission Failure*.

1.3 Detection Scheme Characteristics of Byzantine Failure in Cloud

The Failure detection model should be such that an administrator should be able to monitor Failure data [6]. The model should easily provide up to date failure node data when required by the cloud administrator. A comprehensive failure detection framework is essential for researchers to verify quality of data [6]. Also, the log data of Byzantine Failure should be linked with which it describes, as the self-descriptiveness will ensure that the data is easily comprehended.The goal of this framework is to ensure the quality and trustworthiness of detected Failure[2].

Due to decentralized nature of Cloud computing, traditional methods of collecting failure data is no longer a practical approach[5]. Research needs to be done whether it is possible for cloud administrators to carry out failure detection on their data which is stored on the cloud from a technical standpoint.Architectural idea for a trusted Byzantine Failure tolerant system, and the requirements for effective analyzing of log files of cloud computing systems should be identified for

effective failure detection.

With regard to the increasing need for failure cognition blueprint of open source cloud, this research proposes a scheme that effectively parses through the system log files of the physical machines of cloud environment and collects process activity information which can be used to identify arbitrary failure activities on the cloud. Byzantine failure detection schemes must be capable of detecting execution anomaly and error propagation across multiple machines in the distributed environment, and record the process sequence through which the errors are transferred across vm-instances using socket, providing the opportunity to identify the process tree. This work is capable of achieving the above and hence addresses one of the most recent research issues on cloud computing.

1.4 Rationale of the Study

Availability and performance are key factors for gaining user base in cloud. Fault tolerance of cloud assures both of these features. Byzantine Fault Tolerance is a sub- division of fault tolerance. Hence, by detecting Byzantine fault, a fault tolerance model can be proposed. Thus availability and performance can be assured.

A significant social impact of Byzantine Fault detection of Openstack Cloud can be seen in many educational institutions. Specially, for those who conduct research where extensive computation power and performance is required. Byzantine Fault tolerant system assures real performance without any work flow errors. This work can play a vital role for those systems where precise computation is required without any arbitrary failure.

A notable business impact of capturing Byzantine Fault of Openstack Cloud comes with the growing need of mission critical services among companies. For mission critical compute and storage operations, a robust and error free environment must be assured. If Byzantine Fault of Openstack is detected, it will help

to make cloud more robust. This phenomenon is applicable for companies using Openstack cloud for their business continuity.

Impact of this work cannot be over looked for cloud based information systems also. For case study, the Information Systems under Access to Information (A2I) project of Bangladesh Government can be analysed. This project involves almost all the government driven Information Systems under one hood. Deployment of these infrastructures on cloud can improve service and minimize cost. But, availability and robustness must be assured. Byzantine Fault detection and prevention system can help to make cloud more service oriented in this case. So, Byzantine fault detection of Openstack is of significant importance from this point of view.

From this above discussion, it can be deducted that the service level of cloud computing can be increased to a substantial level by capturing Byzantine faults in the system. Detection of execution anomaly, root cause of error can be identified efficiently if Byzantine fault can be identified. From this captured data decisions can be made easily to ensure performance and to debug the system without down time. Positive result of this study can have impacts on business and social domain. It will also open doors for more future research.

1.5 Research Methodology

The research analyzes the problems for failure detection stated above in the case of open source cloud environments. Since a significant portion of open source cloud platforms are based on Linux, so Linux servers were used for the experiments with open source cloud called OpenStack installed on those. VM-instances were launched using OpenStack and Byzantine Failure detection module installed on physical master servers and nodes of cloud. Figure [4] provides a view of the open source cloud computing environment set up for the experiment. The services shown in Figure[] on the cloud would be monitored during the research and the

various log files will be analyzed to keep track on the activities that occur on the cloud.

The target of the model is to capture the data on the log files (i.e warning and error information) and retrieve those so that those are presented in a readable format. Analysis of existing failure detection protocols for the cloud is necessary to ensure that all the important aspects of cloud during the development of the proposed model are considered. In this experiment, four of Openstack's main modules has been considered namely nova-compute(manage whole service), keystone(manage authentication), neutron(manage network service), glance(manage image service). Dependencies among these services were defined. After that, logs of these services were collected constantly and log keys[9] were made from these log data. Next, the log keys of warning and error log were selected from these data. Then for any warning or error message of a process, the dependent process's log keys were analyzed. Time stamp were also considered to ensure if the error is truly propagating.

The work of Quiang FU et. al. were used as the benchmark. Their work mainly focused on execution anomaly detection. However, it did not state any idea of detecting Byzantine Failure. That is why a standard FSM(Finite State Machine) were defined for the selected process. The execution path of the process call were analyzed against that standard FSM. If the process execute request wrongly rather than stopping, it is assumed that there is a Byzantine Fault in the system.

On the basis of standard benchmarks discussed above, the precision were determined for both the tests and the performances were compared to a set predefined baseline values. Next, the overhead of ByzantineWarrior was detected with respect to the baseline values to obtain a clear understanding of the performance. Upon obtaining the results, the capabilities and drawbacks of ByzantineWarrior

was analyzed and areas where ByzantineWarrior is stronger were identified. Also the frequency and impact of the 10 common Byzantine Error used in the experiment when ByzantineWarrior is installed were provided in a risk matrix. A visual business model was also provided to enable open source cloud service providers to integrate ByzantineWarrior into their system.

Complex processes of exploring large data volumes to determine root cause of error can be simplified if ByzantineWarrior is used in the exploration process. Effective failure detection models allow reusing the log data that are stored on the cloud [1]. This is done by keeping the log of all the processes that are sequentially executed on the cloud using the data set. As a result the data can be reused by repeating the sequence of steps. Thereby, flexibility of data usage can be ensured on the cloud infrastructure.

Due to increased research concentration on performance analysis between various experiments, failure detection can be used to compare the set of steps of one computation with another [10]. Complex data flows of the computational processes can be compared using the failure captured on those, as it provides useful knowledge specifying the detailed methodologies of the two experiments. This information provides empirical data which can be used to compare the performance of the two processes.

A notable social impact of capturing Byzantine failure from cloud log files comes with the increasing demand for information systems. The information systems technologies are becoming greatly dependent on the compute and storage capabilities of the cloud due to their ability to provide these services at comparatively lower cost to traditional infrastructure [11]. With the growing demand for these applications in business organizations, the providers of these applications

are using the cloud infrastructure to store the large volume of data as well as huge computation power for processing it . Due to arbitrary failure, there have been a huge impact on performance[12]. Apart from that, there are miscalculation of processes. Hence the demand for effective techniques to manage such information is highly demanded by cloud architects and administrators. Developing a model based approach to capture Byzantine failure of these data provides cloud administrators with the capability of analyzing the steps through which these failure were produced together with the root cause of error. While the problem is known, it is easier to fix the problem. This yields better management of the information systems computation and data on the cloud.

From the above discussion it can be deduced that the social benefits of cloud computing can be increased to a substantial level by capturing Byzantine failure of cloud computation and storage. The benefits of repeating the computation, minimizing error propagation time, and explore complex experiments to a finer detail by incorporating the fault detection service with the cloud infrastructure. Hence the aim of this research is to analyze existing Byzantine failure detection models and provide a method of capturing Byzantine failure in cloud. The research will have impact of substantial significance and can be used by the companies to better manage their cloud. This research will play important role to make cloud more acceptable to the technology community. It will be a small step towards eradicating the common resource management concerns and social backdrops over cloud computing. This will be achieved by ensuring proper identification of root cause of error, thereby providing greater service management to the authenticated owners.

Chapter 2

Background

In this chapter, the distributed architecture of cloud computing, research assumptions and prerequisites for the research have been analyzed with respect to the research environment. Cloud has enabled many organizations to carry out complex computation with limited resources which would otherwise be impossible [6]. Cloud is an evolving technology, the basis of which is visualization of physical servers to form virtual machines (vm) instances [10]. Despite the advancement in cloud, limitations exist in terms of fault tolerance. However, cloud computing has limitations when it comes to acceptability, accountability, traceability and security. Customers may unwilling to shift to cloud if availability is not ensured. There is a strong requirement to propose effective fault detection mechanisms for this distributed and dynamic environment.

In addition to the conventional Failure problems posed by distributed applications, flexibility, representation and management of arbitrary failure present new challenges for the cloud [13]. Log based failure detection is required to support those. In order to solve these issues, there is an increasing need for research on process-centric and system-centric failure detection from simultaneously generated process logs of cloud computing environment. Arbitrary failure detection will enable administrator to keep track of their performance and availability on the cloud.

Existing mechanisms are not suitable for cloud provenance. Ko. et al identified dependencies among system components as prime challenges for distributed systems [5]. Through real life scenarios of arbitrary failure, the vulnerability of Hadoop and SILK to failure was shown. But the research did not focus on how Byzantine failure detection can effectively prevent such failure. Technical and

procedural approaches to ensure availability of those systems were discussed in[1]. However, the challenges of proposed failure detection system was not shown.

Specific challenges for Byzantine failure detection on the cloud must be addressed. These challenges must highlight log-based failure tracking for process system logs and identification of platform specific failure detection. Visual representations of failure data are an additional challenge. Because of the nature of arbitrary failure, Byzantine Failure occurs in the run time of process execution[9]. Moreover, it is hard to identify the failure because of the hidden characteristics of the failure. Atomic actions at the system level of cloud and critical cloud process files should be identified and studied to find failure vulnerability of those. From analysis of existing research, a rule based failure detection scheme must be proposed for arbitrary error capture and representation.

Byzantine failure detection in distributed system is of increasing importance, as reflected by numerous research works [6],[7],[8]. Arbitrary failure in cloud is different from failure of other distributed system since the failure may have the relationship between the physical machines and the virtual machine instances. This is done to aid cloud administrators identify which physical machine is providing computing and storage resources to which particular virtual machine instance.

2.1 Byzantine Failure

Byzantine Failure occurs in a system when components of a system Fail in arbitrary way(the system does not stop or crush but may process request incorrectly) corrupting their local state and/or producing incorrect or inconsistent outputs[].In standalone systems, Byzantine failure can not propagate to large scale failure.

However, In parallel systems, it often results in avalanche effect. Apart from this scenario, there is no other effective way to detecting this sort of failure.[14]

In most cases effects of Byzantine Failure is hidden. But even in the hidden mode, this failure can cause performance degradation[15]. For services like Cloud, where availability and integrity are key point of concern, detecting Byzantine Failure is of significant importance. So Byzantine Failure can be of two types: omission failure and commission failure. In *Omission Failure* the system halts and stops sending/recieving data. On the other hand, incorrect process request or corrupted request is generated in *commission Failure*.

There is a large body of research on replication techniques to implement highly available systems. The problem is that most research on replication has focused on techniques that tolerate benign faults (e.g., Alsberg and Day [1976], Gifford [1979], Oki and Liskov [1988], Lamport [1989], and Liskov et al. [1991]): these techniques assume components fail by stopping or by omitting some steps. They may not provide correct service if a single faulty component violates this assumption. Unfortunately, this assumption is not valid because malicious attacks, operator mistakes, and software errors are common causes of failure and they can cause faulty nodes to exhibit arbitrary behavior, that is, Byzantine faults. The growing reliance of industry and government on computer systems provides the motif for malicious attacks and the increased connectivity to the Internet exposes these systems to more attacks. Operator mistakes are also cited as one of the main causes of failure [Murphy and Levidow 2000]. In addition, the number of software errors is increasing due to the growth in size and complexity of software.

Byzantine Fault detection can help detecting execution anomaly, work flow error and low performance [2] of cloud. Currently console logs are used to detect such types of error. But, console logs rarely help operators detect problems

in large-scale data enter services. Because, they often consist of the voluminous intermixing of messages from many software components written by independent developers [7]. Though, there are cloud monitoring services like HP Openview, Microsoft SCOM and Amazon Cloudwatch, they do not provide monitoring schemes for Byzantine fault [16]. Moreover, these applications work in a centralized way making them prone to single point of failure []. Hence a model for extracting execution anomaly detection for cloud must be addressed [7].

Cloud administrators need to keep track of the process workflow and anomalistic behaviour to keep cloud system up and running. Hence, every unusual log reports (warning and errors) must be tracked for fault detection manually []. Moreover, desired performance benchmark must be obtained for production level cloud.

2.2 Assumptions

The research is taken forward while keeping in mind certain assumptions and obstacles regarding cloud computing failure detection. The first two are related to availability, the next two are related to adoption and the final two are derived from regulations and financial backdrops. Each problem is paired with an opportunity showing how Byzantine failure detection can help solve the issues, ranging from failure detection to analysis of the captured log data.

2.3 Process Availability and Audibility

Availability of the cloud is often the most cited objection since customers are severely concerned about the availability of the service data they store on the

cloud. None the less, they are concerned about data loss. The availability issues which are used to protect the cloud from down time are similar to the ones implemented at large scale data centers, with the exception that both customers and not only cloud administrators are responsible for the availability. In addition to the two parties, a third party may be involved for the availability and robustness of the cloud as well which includes providers of value added services which are incorporated into the cloud. For example, RightScale provides value added services, a few of which are automatically incorporated with Amazon Elastic Cloud Compute (EC2) and helps dynamic scale up or scale down of EC2.

The cloud users are responsible for the availability of the vm instances which are sanctioned. The cloud administrators are responsible for the availability of the physical layers, as well as the availability of vm layer by monitoring data regarding launching of vm instances and tracking logs of file access and changes made to those. So availability of the intermediate layers and vm instances are shared among the cloud administrators and users. If the users are exposed to greater details of vm instances, more responsibilities are handed over from the administrator to the user. Some cloud providers outsource the availability maintenance task to third parties who have independent IT management to manage the cloud services. Amazon EC2 users have more technical responsibility than Azure users, who in turn have more technical responsibilities than AppEngine users [9].

The primary Byzantine fault detection mechanism that can be used in todays cloud infrastructure is log based detection. Log based failure detection can be used to monitor the vm instances from launching to termination [9]. This is a powerful tool to ensure confidentiality and audibility, since error that may occur in future can be identified from the warning stage. However, failure detection must be based on log analysis and it is not bug free as certain safe actions can be

considered as potential failure.

Incorrect failure detection mechanisms can result in unwanted process restart and system overhead. The challenges are similar to large data centers which do not implement cloud computing, where various programs need to be co-operated from each other physically. Any large data center, be it cloud or non-cloud, will look to detect arbitrary failure of the physical layer as well as vm layer to ensure services.

2.4 Log Key Extraction

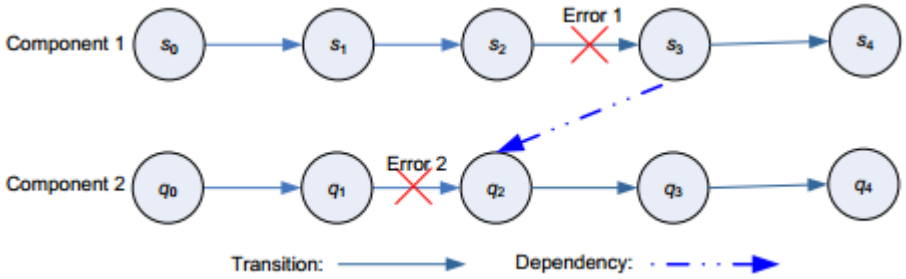
Systems logs usually record run-time program behaviors, including events, states and inter-component interactions. An unstructured log message often contains two types of information: one type is free-form text string that is used to describe the semantic meaning of a recorded program behavior; the other type is a parameter that is used to express some important system attributes. In general, the number of different log message types is often huge or even infinite because of various parameter values. Therefore, during log data min-ing, directly considering log messages as a whole may lead to the curse of dimension.

In order to overcome this problem, each log message was replace by its corresponding log key to perform analysis. The log key is defined as the common content of all log messages which are printed by the same log-print statement in the source code. In other words, a log key equals to the free-form text string of the log-print statement without any parameters. For example, the log key of log message 5 (shown in Figure 1) is **Image file of size saved in seconds**. Logs were analyzed based on log keys because: (1) In general cases, different log-print

statements often output different log text messages. It means that each type of log key corresponds to one specific log-print statement in the source code. Therefore, a sequence of log keys can reveal the execution path of the program. (2) The number of log key types is finite and is much less than the number of log message types. It can help to avoid the curse of dimension during data mining.

The challenging problem is that neither which log messages are printed by the same log-print statement nor where parameters are in log messages is known. Therefore, it is very difficult to identify log keys. Generally, the log messages printed by the same statement are often highly similar to each other, while two log messages printed by different log-print statements are often quite different. Based on this observation, clustering techniques to group log messages printed by the same statement together were used, and then their common part were used as the log key.

However, the parameters may cause some clustering mistakes because the log messages printed by different statements may also be similar enough if they contain a lot of identical parameter values. In order to reduce the parameters influence on clustering, we first erase the contents that are obvious parameter values according to some empirical knowledge. Then, we further apply a raw log key clustering and group splitting algorithm to obtain log keys. **Figure** gives an example to illustrate the procedure of extracting log keys from log messages. The full method of log key extraction is described in **chapter 4**. This model is influenced by the work of et.al [5].



2.5 Root Error Localization

In many distributed systems, an error occurring at one component often causes execution anomalies of other components due to inter-component dependencies. Therefore, a group of related errors from different system components are often detected at the same time. Figuring out the error propagation path and locating the root error site (not root cause) become an important step of problem diagnosis. In this section, an approach to find the relationship among a set of related errors using the learned inter-component dependencies is described.

The basic idea of error localization is illustrated in 2.5. In a normal execution of Component 1, Component 1 prints log keys from s_0 to s_4. Similarly, Component 2 prints log keys from q_0 to q_4. In an error case, Error 1 and Error 2 are detected in Component 1 and Component 2, since a transition from s_2 to $s_3$and a transition from n_1 to $_2$ cannot happen.s_3 and q_2 is called **inaccessible** log of Error1 and Error2 respectively. Because log item q_2 of Component2 is dependent on log item s_3 of Component1, Component2 cannot print out q_2 if Component1 does not print s_3. Therefore, given Error1 and Error2, it can be concluded that Error1 and Error2 are related and that Error1 causes Error2 in maximum likelihood. This shows the simple and basic idea behind our root error localization approach.

2.6 Attributes of Byzantine Fault Detection Systems for the Cloud

ByzantineWarrior for the cloud must ensure that the execution time, execution path and anomaly behavior are correctly identified. Following are some essential attributes which satisfy the stated capabilities.

1. **Process state Coupling:** property states that a process and its failure data must match that is, the log data must accurately and completely describe the process status. This property allows users to make accurate decisions using log keys. Without process coupling, a process might reckon error that actually originated from other process. In this case, the user detection system may lead to misleading detection. The main point is to identify whether the error originated from the very process or procreated from another process.

2. **Causal Ordering:** property acknowledges the causal relationship among processes. If a process δ_p is the result of propagating error from η_p, then the failure of δ_p is the subset of the failure of η_p. Thus, a system must ensure that a process's ancestors (and their failure) are persistent before making the process itself persistent. Multi-process Causal Ordering violations occur when the system detects a process failure to before detecting all its ancestors, and the system crashes before recording those ancestors and their failure data. Similar to eventual data coupling,a weaker form of the property Eventual Causal Ordering is realizable. A system still requires detection to account for the intervals during which a process failure may be incomplete, because its ancestors and their error are not yet persistent or not available due to eventual consistency.

3. **Efficient Query:** Since various log message is created more frequently than

it is queried, efficient error recording is essential. However, efficient query is also important as error message must be accessible by the Byzantine Failure . In scenarios where the number of processes is few , efficiency is not an issue. Efficiency matters, however, when the number of processes is sizeable and the system is unsure of the processes that it want to access.

4. **Rate of False Positives (FP):**High rates of FP will result in increased workload in analyzing and responding to events. Theymay also result in reduced productivity due to the prevention of legitimate documents and messages from reaching employees.

5. **Rate of False Negatives (FN):**As with other security measures, a high rate of false negatives will lead to a false sense of security, plus potentially placing the organization in jeopardy from confidential data which is leaked without being identified. At the same time, provenance data should have the ability to detect data flooding, file type/format manipulation. ...

2.7 Conventional Log Based Fault Detection Limitations

ByzantineWarrior can be widely used in failure detection of the cloud to identify the root cause of system failures. The cloud should be available 24*7. However even reputed cloud service providers like Amazon EC2 and Google AppEngine have faced service breaks. On April21,2011 the service of Amazon EC2 gone down apparently with out no reason.[11] After 5 days of trouble shooting, finally they found out that an anomaly behavior of a sub process of compute node triggered an arbitrary failure that resulted in a system crash. With effective arbitrary failure detection from logs, data forensic experts can successfully determined the reason of the flaws. Following table illustrates some of the service disruptions of proprietary

Service	Reason of Failure	Down time
Amazon S3	One of the Compute node triggered Arbitrary Fault	52 Minutes
Google AppEngine	Error from the end of the maintenance engineers program	4.6 Hours
Gmail	The contact list mechanism crashed	1.4 Hours

Table 2.1: Down time of various Services due to Failure

cloud services and the reasons identified for such failures [10].

2.8 Critical Scheduling of Cloud Systems

The scheduler maps the nova-API calls to the appropriate OpenStack components. It runs as a daemon named nova-schedule and picks up a compute server from a pool of available resources depending on the scheduling algorithm in place.

1. **No prior knowledge about incoming processes:** A good process scheduling algorithm of distributed systems should work with no prior knowledge of the processes or operations which are to be executed. Algorithms which function based on prior in-formation pose an additional burden on the users since the users then must explicitly specify the characteristics of the operation they submit.

2. **Dynamic load balancing capability:** The algorithm should have the capability of managing the dynamically changing load of the nodes. Process assignment on different nodes should be based on the current load status of the nodes and not on some pre specified policy.

3. **Thrashing Resistant:** Processor thrashing occurs when the algorithm causes all the nodes of the system to spend all their time migrating processes without accomplish-ing any useful work

4. Scalable: The algorithm should be capable of supporting a large number

of nodes from a small number without computational complexity. The attribute must sustain even when the number of nodes is suddenly reduced.

5. **Fault Tolerant:** The algorithm should not suffer from crash of a single node of the system. The system should continue functioning with the available nodes which are up at that point of time. The cloud controller should be capable of provisioning resources from its own physical machines which would increase the fault tolerance capability of the system.

6. **Quick decision making:** The scheduling algorithm must exhibit good decision making capability. This is very important at times when the computational requirements of the processes are very high. For example, a customer executing scientific calculations on the instances may require sudden requirement of storage. The algorithm must be able to provide the required storage without termination of the instances.

A scheduler can base its decisions on various factors such as load, memory, physical distance of the availability zone, CPU architecture, etc. The nova scheduler implements a pluggable architecture. Currently the nova-scheduler implements a few basic scheduling algorithms

1. **Chance:** In this method, a compute host is chosen randomly across availability zones.

2. **Availability zone:** Similar to chance, but the compute host is chosen randomly from within a specified availability zone.

3. **Simple:** In this method, hosts whose load is least are chosen to run the instance. The load information may be fetched from a load balancer.

2.9 Load Balancing approach of the resource scheduler

In this approach all the processes submitted by the user are distributed among the nodes of the system so as to equalize the workloads among the nodes. Another approach is the load sharing approach in which the target is to conserve the ability of the system to perform work by assuring that no node is idle while processes wait for being processed. Following are a number of desirable features for the distributed environment scheduler of the cloud.

2.10 Message passing mechanisms in distributed environment

Traditional interprocess communication (IPC) systems contain protocols developed on the message passing model. Despite the protocols ability to support distributed applications, their flexibility remains a lot to be desired [11]. As a result applications of heterogeneous types are not supported in such IPC systems. Remote Procedure Call (RPC) provides a communication mechanism that solves this problem. This mechanism can support distributed applications with heterogeneous attributes.

In this chapter, a developed distributed application that consists of taking two integers as input at the client then sending the numbers over to the server and using the RPC model and executing the mathematical equation.

The report show how RPC mechanism is implemented on the basis of send and receive primitives for communication with a disjoint machine. It has also been identified that how various components of RPC can be used to satisfy the applications requirements. Also the importance of RPC elements to abstract the internal

complexity of message passing from the user is shown. The RPC mechanism used to implement this application consists of the RPC Runtime, that hides the underlying network details from the end user. Five important elements of the RPC technique used here are: Client process, Client stub, RPCRuntime, Server process and Server stub.

1. *At Client:*

    - A call is generated from the client process to the client stub
    - The client stub packs the request with the correct parameters and sends it over to the RPCRuntime for transferring it to the server
    - The RPCRuntime transfers the message from the client to the server machine over the network

2. *At Server:*

    - The RPCRuntime of the server receives the message. The message is passed to the server stub
    - The message is received and unpacked at the serve stub. The stub invokes the call to the desired server process
    - On receiving the unpacked message, the required actions are executed by the server process and the results are forwarded to the server stub.

3. *Attributes of the application:* The application is based on stateless server mechanism. The moment the connection is lost, the server does not keep track of the client machine. At the same time the parameter arguments are called by reference. As a result of this application the server and clients are on the same network.

4. *Limitations:* The application can be extended to a great length. The application will not work on remote machines outside the network of the server.

The client executable must be run each time a pair of numbers are given as input

2.11 Rabbit MQ Server

OpenStack communicates among themselves using themessage queue via AMQP (Advanced Message Queue Protocol). Nova uses asynchronous calls for request response, with a call-back that gets triggered once a response is received. Since asynchronous communication is used, none of the user actions get stuck for long in a waiting state. This is effective since many actions expected by the API calls such as launching an instance or uploading an image are time consuming.

2.12 Platform Dependency

Open source cloud requires detailed understanding, whereas proprietary clouds have closed source. The aim of this research is to develop a Byzantine Failure detection model for the cloud that is independent of any platform and fits into any cloud environment. Since large number of processes are prone to Byzantine Failure, research should be carried to present this detection result in a visual format to the viewers . Visual representations of Failure Detection must be specific for the cloud customers, service providers and regulators since their needs are different. Hence research should be carried out in this regard.

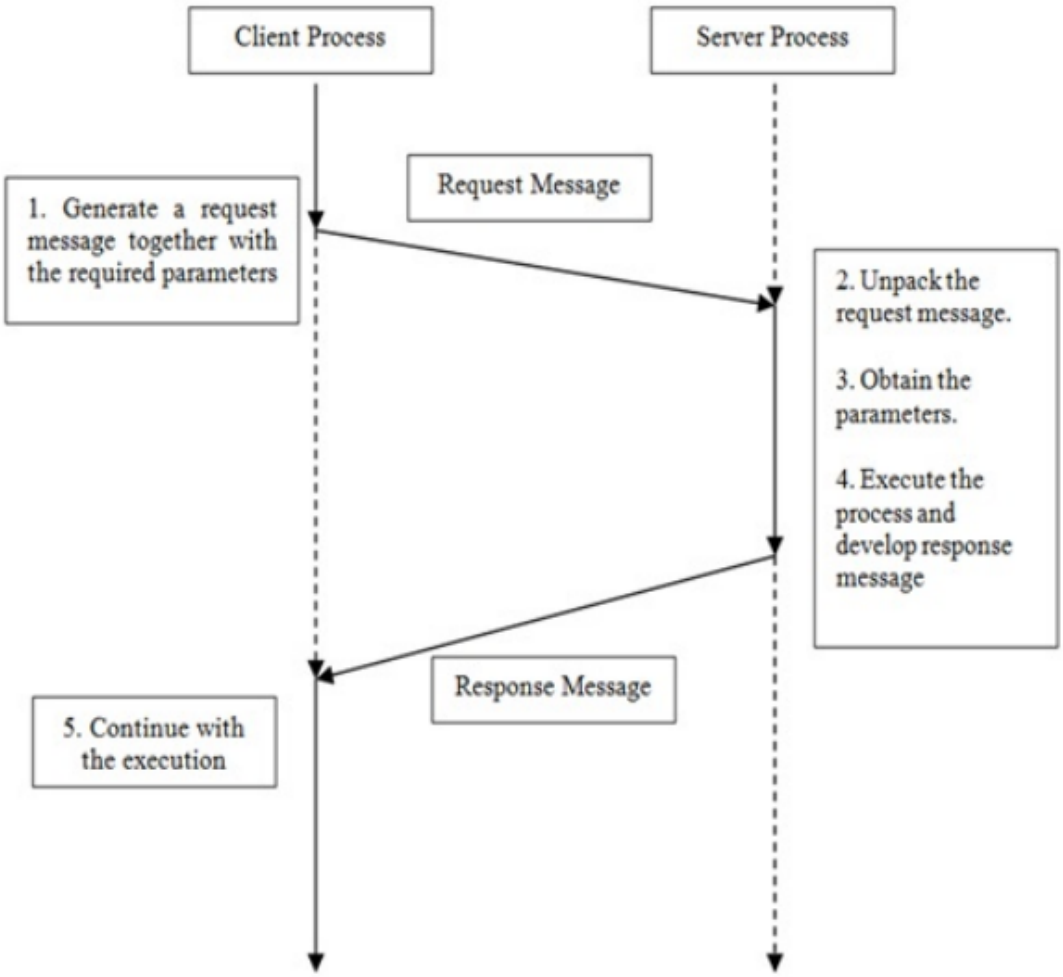


Figure 2.1: Architecture of Rabbit MQ server

Chapter 3

Literature Review

In this chapter, the research issues in Byzantine failure detection have been highlighted. The chapter focuses on the existing research work that has been carried out regarding fault detection. At the same time the scope of research in this growing field are highlighted. With regard to the rapid growth of cloud, effective failure detection will ensure that the system is robust and fault tolerant. Byzantine failure based on system and operation logs will ensure the integrity of that system. Identification of failure attributes and obstacles to failure detection in the cloud is necessary for user friendliness and effectiveness of failure detection. Also research in the field of fault detection will ease cloud maintainability and at the same time develop user confidence. Preliminary findings of the computer forensic community in the field of digital forensics have to be revised and adapted to then environment. Investigators need the possibility of reconstructing the corresponding environment for recreating scenarios and test hypothesizes. Hence the scope of work in this area is manifold.

3.1 Fault Tolerance in Cloud

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. Ko.et.al. [17] 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.

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 watch dogging, checkpointing and logging critical processes have not been considered for the cloud.

3.2 Byzantine Fault Tolerance(BFT)

BFT is an algorithm for state machine replication [Lamport 1978; Schneider 1990] that offers both liveness and safety provided at most $\lfloor (n-1)/3 \rfloor$ out of a total of n replicas are faulty. This means that clients eventually receive replies to their requests and those replies are correct according to linearizability [Herlihy and Wing 1987; Castro and Liskov 1999a]. BFT is the first Byzantine-fault-tolerant, state machine replication algorithm that is safe in asynchronous systems such as the Internet: it does not rely on any synchrony assumption to provide safety. In particular, it never returns bad replies even in the presence of denial-of-service attacks. Addition-ally, it guarantees liveness provided message delays are bounded eventually. The service may be unable to return replies when a denial-of-service attack is active but clients are guaranteed to receive replies when the attack ends. Since BFT is a state machine replication algorithm, it has the ability to replicate services with complex operations. This is an important defense against

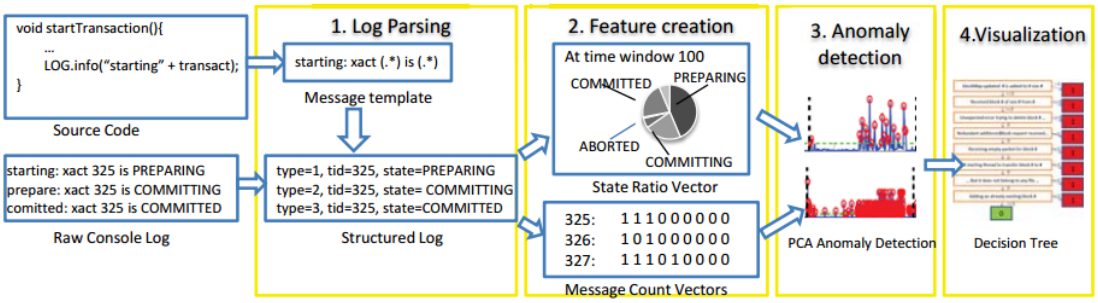


Figure 3.1: Workflow Based approach of Fault Detection

Byzantine-faulty clients: operations can be designed to preserve invariants on the service state, to offer narrow interfaces, and to perform access control. BFT provides safety regardless of the number of faulty clients and the safety property ensures that faulty clients are unable to break these invariants or bypass access controls. Algorithms that only offer reads, writes, and synchronization primitives (e.g., Malkhi and Reiter [1998b]) are more vulnerable to Byzantine-faulty clients; they rely on clients to order and synchronize reads and writes correctly in order to enforce invariants.

3.3 Workflow Based Fault Detection

In this approach, the workflow of a process execution is considered to be the blue print of process behavior. First, the typical workflow of a process is determined from the console logs. Next, a standard FSA is created. After that the running process flow is compared to the standard process flow. The process is shown in the figure 3.1 This is done in four steps:

1. *Log parsing:* first convert a log message from unstructured text to a data structure that shows the message type and a list of message variables in (name, value) pairs. All possible log message template strings from the source code and match these templates to each log message to recover its

structure (that is, message type and variables). These experiments show that we can achieve high parsing accuracy in real-world systems.

There are systems that use structured tracing only, such as BerkeleyDB (Java edition). In this case, because logs are already structured, we can skip this first step and directly apply our feature creation and anomaly detection methods. Note that these structured logs still contain both identifiers and state variables.

2. *Feature creation:* Next, feature vectors is construct from the extracted information by choosing appropriate vari-ables and grouping related messages.This metoh focuses on constructing the state ratio vector and the message count vector features, which are unexploited in prior work.

3. *Anomaly detection:* Then, anomaly detection methods were applied to mine feature vectors, labeling each feature vector as normal or abnormal. It was found that the Principal Component Analysis (PCA)-based anomaly detection method [18] works very well with both features. This method is an unsupervised learning algorithm, in which all parameters can be either chosen automatically or tuned easily, eliminating the need for prior input from the operators.

4. *Visualization:* Finally, in order to let system integrators and operators better understand the PCA anomaly detection results,results were visualized in a decision tree[19]. Compared to the PCA-based detector, the decision tree provides a more detailed explanation of how the problems are detected, in a form that resembles the event processing rules [10] with which system integrators and operators are familiar.

3.4 Rule Based Fault Detection

In rule based fault detection, some protocols are applied to a process execution life cycle in respect of time, log parameters and log keys. For a log message m,if extracted log key is $K(m)$, numbers of parameters are $NP(m)$ and the i^{th} parameter's value as $PV(m.i)$, each log message with time stamp $T(m)$ can be represented by a multi-tuple $[T(m), K(m), PV(m,1), PV(m,2)....., PV(m, PN(m))]$. The rule is determined in four steps:

- Identification of Related Log Key Pairs
- Determine the dependency direction
- Variance based False Positive Reduction
- Dependency Pruning

because of the huge amount of log data needed to be calculated, Rule based Fault detection in cloud shows weaker results in respect of other approaches like workflow based model.

3.5 Feature Based Fault Detection

In this method, the log messages are cross matched according to the features they describe about the process. Feature elaborates the characteristics of the processes in a better way than rule based or workflow based models. This process is the hybrid of rule based and workflow based model. In Feature based fault detection, processes are relate through the probability of processes matching their feature. The main components of this skim are:

- **Erasing parameters by empirical rules** is done by following Rule based model.

- **Raw log key clustering** is done with the help of workflow based model.
- For **Group splitting**, Feature based method has own algorithm
- **Determine log keys for new log messages** are done in the run time

3.6 Byzantine Fault Attributes

Byzantine Fault attributes involve the prerequisites that are necessary for successful fault detection in a distributed and ubiquitous computing environment like the cloud [15].Wei Xu et al identified key attributes for fault detection on the cloud [20]. The key attributes include fault data collection querying from console logs. Another attribute is the multi-process causal ordering property which acknowledges the causal relationship among processes and ensures that an process's ancestors are persistent before making the process persistent itself [21]. A notable attribute is the data-independent persistence property which ensures that a system retains an process's failure data, even if the process is killed. These attributes are discussed below to a greater detail.

1. **Process state Coupling:** property states that a process and its failure data must match that is, the log data must accurately and completely describe the process status. This property allows users to make accurate decisions using log keys. Without process coupling, a process might reckon error that actually originated from other process. In this case, the user detection system may lead to misleading detection. The main point is to identify whether the error originated from the very process or procreated from another process.
2. **Causal Ordering:** property acknowledges the causal relationship among processes. If a process δ_p is the result of propagating error from η_p, then the failure of δ_p is the subset of the failure of η_p. Thus, a system must ensure that

a process's ancestors (and their failure) are persistent before making the process itself persistent. Multi-process Causal Ordering violations occur when the system detects a process failure to before detecting all its ancestors, and the system crashes before recording those ancestors and their failure data. Similar to eventual data coupling,a weaker form of the property Eventual Causal Ordering is realizable. A system still requires detection to account for the intervals during which a process failure may be incomplete, because its ancestors and their error are not yet persistent or not available due to eventual consistency.

3. **Efficient Query:** Since various log message is created more frequently than it is queried, efficient error recording is essential. However, efficient query is also important as error message must be accessible by the Byzantine Failure . In scenarios where the number of processes is few , efficiency is not an issue. Efficiency matters, however, when the number of processes is sizeable and the system is unsure of the processes that it want to access.

4. **Rate of False Positives (FP):**High rates of FP will result in increased workload in analyzing and responding to events. Theymay also result in reduced productivity due to the prevention of legitimate documents and messages from reaching employees.

5. **Rate of False Negatives (FN):**As with other security measures, a high rate of false negatives will lead to a false sense of security, plus potentially placing the organization in jeopardy from confidential data which is leaked without being identified. At the same time, provenance data should have the ability to detect data flooding, file type/format manipulation. ...

3.7 Log Based Fault Detection Obstacles

ByzantineWarrior can be widely used in failure detection of the cloud to identify the root cause of system failures. The cloud should be available 24*7. However even reputed cloud service providers like Amazon EC2 and Google AppEngine have faced service breaks. On April21,2011 the service of Amazon EC2 gone down apparently with out no reason.[] After 5 days of trouble shooting, finally they found out that an anomaly behavior of a sub process of compute node triggered an arbitrary failure that resulted in a system crash. With effective arbitrary failure detection from logs, data forensic experts can successfully determined the reason of the flaws.

1. *Business Continuity and Service Availability:* Technical issues of availability aside, a cloud provider could suffer outages for nontechnical reasons, including going out of business or being the target of regulatory action [22].

2. Data Lock-In: Concern about the difficult of extracting data from the cloud is pre-venting some organizations from adopting Cloud Computing. Customer lock-in may be attractive to Cloud Computing providers

3. *Data Confidentiality and Audibility:* In special cases when the source of failure cannot be bound to a specific entity, procedures will be carried out to decide the violating entity in a best effort manner. The system shall suffer minimal delay due to dispute resolution [25].The security issues involved in protecting clouds from outside threats are similar to those already facing large data centers, except that responsibility is divided among potentially many parties, including the cloud user, the cloud vendor, and any third party vendors whose value-added services have been bundled into the cloud offering [23].

4. *Performance Unpredictability:* The obstacle to attracting HPC is not the

use of clusters; most parallel computing today is done in large clusters using the message-passing interface MPI. The problem is that many HPC applications need to ensure that all the threads of a program are running simultaneously and today's virtual ma-chines and operating systems do not provide a programmer-visible way to ensure the above statement [24].

5. *Reputation Fate Sharing:* One customers bad behavior can affect the reputation of the cloud as a whole. For instance, blacklisting of EC2 IP addresses [25] by spam prevention services may limit which applications can be effectively hosted.

To address the above issues, a methodology was adopted. The methodology showed that the hybrid method of detecting fault in distributed systems like cloud is required to address the Byzantine Fault.

Chapter 4

ByzantineWarrior: A Byzantine Fault Detection Framework

As discussed in the previous chapters, a summary of the cloud log based detection can be drawn on the ground of the limited work on fault detection for a highly complex and distributed environment such as open source clouds like OpenStack [26]. The chief reason for this is the dynamic nature of cloud environments. The lack of trust on the cloud service providers from the customers perspective poses a big risk to the success of the cloud [22]. The principal reasons behind this are the limited availability of log based fault detection tracking for open source cloud.

4.1 Proposed Model for Byzantine Fault Detection

According to current scenarios, there is no effective way to detect Byzantine Fault in the application layer of Openstack cloud. Finding a way to detect these sorts of faults in Openstack is very important to ensure availability and performance. This work will concentrate on an effective way of detecting Byzantine faults of cloud's system modules. Log analysis mechanism will be used as the method of identification.

In a nutshell, our research aims to find a generalized model for detecting Byzantine fault on the cloud using log data. The recently developed research in execution anomaly detection by log analysis [9], Scientific Workflow error detection and scheduling [2], Automated Error Detection using FSA[5] have been applied to a very specific extent. Their works, therefore, need to be investigated critically.

	Obstacle	Opportunity
1	Availability of Service	Use Multiple Cloud Providers; Use Elasticity to Prevent DDOS
2	Data Lock-In	Standardize APIs; Compatible SW to enable Surge Computing
3	Data Confidentiality and Auditability	Deploy Encryption, VLANs, Firewalls; Geographical Data Storage
4	Data Transfer Bottlenecks	FedExing Disks; Data Backup/Archival; Higher BW Switches
5	Performance Unpredictability	Improved VM Support; Flash Memory; Gang Schedule VMs
6	Scalable Storage	Invent Scalable Store
7	Bugs in Large Distributed Systems	Invent Debugger that relies on Distributed VMs
8	Scaling Quickly	Invent Auto-Scaler that relies on ML; Snapshots for Conservation
9	Reputation Fate Sharing	Offer reputation-guarding services like those for email
10	Software Licensing	Pay-for-use licenses; Bulk use sales

Figure 4.1: Top 10 Obstacles to and Opportunities for Growth of Cloud Computing

Once the analysis is completed, individual but primitive entities will be identified, with the view of develop a more generalized protocol, which could be applied in broader extent.

As mentioned above, once the log based Byzantine fault detection framework is available for OpenStack, an organizational business model needs to be proposed that would identify the core members of the cloud team within a company who are involved or are affected by this fault detection scheme [27]. This is necessary as it will help cloud providers identify the players involved in the implementation, operation and maintenance of the cloud service and also define the auditability and accountability relationships with other cloud providers and aid in the determination of Service Level Agreement (SLA) with customers as well.

A mathematical model with effective performance analysis results must be presented to identify the process trees in the cloud and evaluate modifications to system critical processes for better detection of Byzantine failure in open source cloud environments. Such evaluations have been done to a minimal extent for open source cloud, the reason being the organizational dependencies on proprietary cloud services [28]. However with the shift in the interest of business organizations towards open source cloud platforms [25], such models with analysis has become a must need. According to Armbrust et al, there are 10 top challenges to be cloud a

success (see Figure 4.1)[6]. The proposed model can solve availability, bug fixing and performance issues to a great extant. Hence the framework can contribute to the mitigation of majority of threats in open source cloud environments.

4.2 Log Key Extraction

In the previous chapter **??**, raw log key generation is described to some extent. In this section, the total procedure will be described. After raw log key generation, clustering log keys according to their features and group splitting are other two challenges. The method is described below.

Before clustering, a proper metric to represent the similarity of two raw log keys must be found out. The string edit distance is a widely used metric to represent the similarity between word sequences. It equals to the number of edit operations required to transform one word sequence to the other. One edit operation can operate only one word. The operation can be adding, deleting or replacing. Obviously, the edit distance only counts the number of operated words; it does not consider the positions of the operated words. However, for this problem, the positions of operated words in raw log keys are meaningful for measuring similarity. It is because most programmers tend to write text messages (log keys) firstly, and then add parameters afterwards. Therefore, words at the beginning of raw log keys have more probability to be parts of log keys than words at the end of raw log keys do. Therefore, the operated word at the beginning of the raw log keys should be more significant for measuring raw log keys difference. Based on this observation, we measure raw log keys similarity by the weighted edit distance, in which we use sigmoid similar function to compute weights at different positions.

For two raw log keys rk_1 and rk_2 , the necessary operations required to transform rk_1 to rk_2 is denoted as $\theta_1, \theta_2, ..., \theta_\phi$; ϕ is the is the number of neces-

sary operations. The weighted edit distance between rk_1 and rk_2 is denoted as $WED(rk_1, rk_2)$,

$$WED(rk_1, rk_2) = \sum_{i=1}^{\phi} \frac{1}{1 + e^{x_i - v}}$$

Here,x_i is the index of the word that is operated by the i^{th} operation OA_i ; $v$is a parameter controlling weight function.

Similar raw log keys are clustered together. For every two log keys, if the weighted edit distance between them is smaller than a threshold Δ , they are connected with a link. Then, each connected component corresponds to a group which is called as an initial group. The initial group examples are shown in the third block in Figure 4.2.

4.3 Defining Finite State Machine(FSM)

In order to detect anomalies of work flows, a Finite State Automaton (FSA) is used to model the execution behavior of each system module. Although there are some other alternate models, such as Petri-Net, we adopt FSA because it is simple but effective. FSA has been widely used in testing and debugging software applications [29]. A FSA consists of a finite number of states and transitions between the states. A set of algorithms have been proposed in previous literature to learn FSA from sequential log sequences [6]]. Here the algorithm used was proposed by [16] to learn a FSA for each system component from training log key sequences which are produced by normally completed jobs. Each transition in the learned FSAs corresponds to a log key. All training log key sequences can be interpreted by the learned FSAs. Therefore, each training log key sequence can be mapped to a state sequence.

State	Interpretation
S87 S96	Initialization when a new job submitted
S197	Add a new map/reduce task
S103	Select remote data source
S99	Select local data source
S198	Task complete
S106	Job complete
S107	Clear task resource

Figure4.3 shows the example of the learned FSM of JobTracker of Hadoop []. The state interpretations according to the log messages given in Table **??** . From the learned the FSM, the following work flow is obtained: from S87 to S96, the JobTracker carries out some initialization tasks when a new job is submitted. After initialization, the state machine enters S197 to add a new Map/Reduce task. For each map task, it selects local or remote data source for processing. Then, the task is completed. When the last task is finished, the job is completed, and all resources of tasks are cleared iteratively. In fact, the learned FSM correctly reflects the real work flow of the JobTracker.

4.4 Description of the Proposed Framework

The algorithm 1 refers to the main workflow of ByzantineWarrior. Here, P is the set of all process in a module. S defines the i^{th} process's states,counter (κ) counts the number of faulty states and number of states in i_{th} process n_{p_i}. First, the framework reads each process in queue. Then it checks for any faulty states by cross matching the standard state defined in S. If the process have a faulty state it increase the counter. A threshold value is set for the detection of Byzantine fault. For traditional Byzantine problem the value is $3\kappa + 1$. It is the trivial value for which Byzantine Failure may be roll over. But, for real time use, the threshold is set to $2\kappa + 1$, because standard industrial practice define 70 percent error as a

sure fault.

Data: set of Processes(P),Set of states of p_i (S), counter (κ),number of states in i_{th} process n_{p_i}

Result: Process p_i that has Byzantine Fault

initialization;

while $P \leftarrow p_i$ **do**

  read read all process id PID;

  if s_i^r *not equal* s_i **then**

    increase κ by 1;

    if n_{p_i} *is greater than* $2\kappa + 1$ **then**

      return p_i;

    else

      go back to the beginning of current section;

    end

  else

    go back to the beginning of current section;

  end

end

Algorithm 1: ByzantineWarrior Fault Detection Algorithm

4.5 Extension of the Openstack Model

The implementation and performance analysis of the proposed fault detection model for system administration have been implemented and tested on OpenStack Essex cloud running in Ubuntu 12.04 servers. Hence the OpenStack model has been extensively analyzed and a proposition to integrate the proposed fault detection framework has been presented to increase the security of OpenStack and increase its acceptability to the business world.

The extension of the OpenStack model will work in line with a number of mod-

ules of the existing OpenStack structure which includes collecting log data from Openstack scheduler process, Image register and upload process, nova-compute and nova-network processes respectively. Information regarding process execution are also collected by the proposed framework. This will enable OpenStack cloud administrators to keep track of the cloud and detect anomalistic execution in the cloud computing infrastructure.

Figure 4.4 shows an overview of the extended OpenStack model and shows how the extended part of the framework gathers information from the existing cloud modules with-out hindering the activities of those. The following is a description of the processes with which the proposed framework communicates and collects process execution information.

1. **Nova-schedule**: The nova-scheduler in OpenStack is used to schedule the events that take place in the cloud such as the event of launching a vm instance [2]. The frame-work collects information about the schedule and feeds the information to the cloud administrator thereby assisting the administrator in the decision making process that a vm-instances has been launched successfully.

2. **Image upload and register**: The process determines whether an image of a operating system has been uploaded and once uploaded, whether the image has been registered and a unique registration id has been provided that will enable the users to launch in-stances of that image [30]. The framework collects information from the process and enables the cloud administrator to identify the time, date, version and permissions of the uploaded and registered image.

3. **Nova-compute:** The journals of nova-compute are monitored by the proposed frame-work to provide information about critical proceedings on the cloud such as allocation of virtual network, the resource allocation from

physical machines.

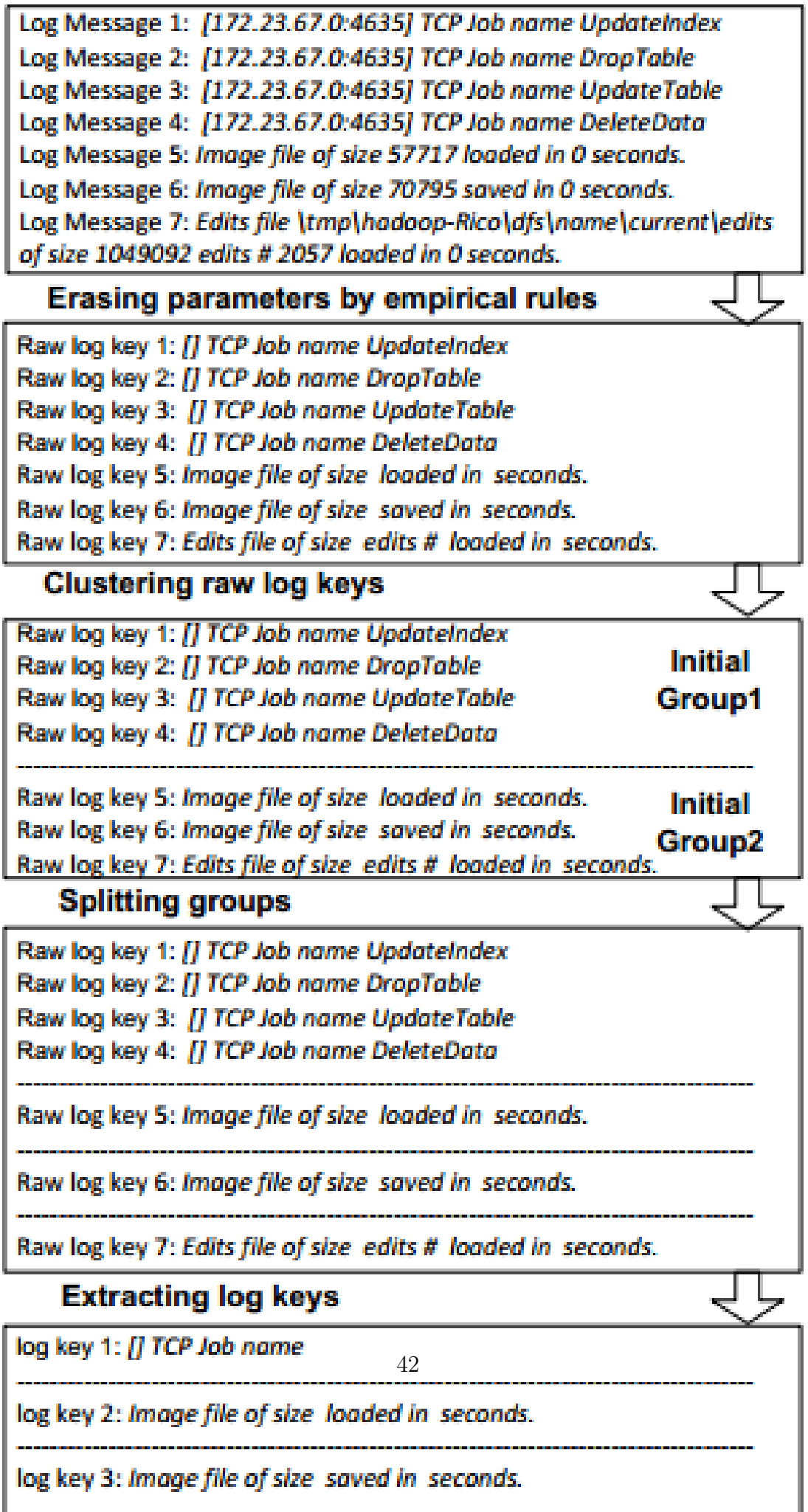
4.6 Solution of the addressed problems by the proposed model

The problems stated in earlier section of this chapter can be solved through the Byzantine Fault detection scheme proposed in this research. The highlighted problems with their solutions are stated below.

1. Limited work on Byzantine Fault detection for open source cloud: It was stated in the related work chapter that research on Byzantine Fault detection for open source cloud has been conducted to a limited extent [31], [14], [32]. The chief reason for this being that most research are funded by proprietary cloud service providers and hence the test bed are the proprietary cloud platforms. This research puts forward a log based Fault detection model for OpenStack which is open source. Therefore the research contributes in the fault detection field for open source cloud computing therefore helping open source cloud administrators better manage the open source cloud platforms and collect data for investigation.

2. Lack of Availability: Availability is one of the main concerns of Cloud computing[13]. Through this work, a novel mechanism of detecting arbitrary failure is discussed.

3. Performance Issues: The main adverse effect of Byzantine Failure in Distributed Systems is that it hinders the performance of the system to a large scale. With this work, a method of detecting such failure is highlighted.


Log Message 1: [172.23.67.0:4635] TCP Job name UpdateIndex
Log Message 2: [172.23.67.0:4635] TCP Job name DropTable
Log Message 3: [172.23.67.0:4635] TCP Job name UpdateTable
Log Message 4: [172.23.67.0:4635] TCP Job name DeleteData
Log Message 5: Image file of size 57717 loaded in 0 seconds.
Log Message 6: Image file of size 70795 saved in 0 seconds.
Log Message 7: Edits file \tmp\hadoop-Rico\dfs\name\current\edits of size 1049092 edits # 2057 loaded in 0 seconds.
Erasing parameters by empirical rules
Raw log key 1: [] TCP Job name UpdateIndex
Raw log key 2: [] TCP Job name DropTable
Raw log key 3: [] TCP Job name UpdateTable
Raw log key 4: [] TCP Job name DeleteData
Raw log key 5: Image file of size loaded in seconds.
Raw log key 6: Image file of size saved in seconds.
Raw log key 7: Edits file of size edits # loaded in seconds.
Clustering raw log keys
Raw log key 1: [] TCP Job name UpdateIndex
Raw log key 2: [] TCP Job name DropTable
Raw log key 3: [] TCP Job name UpdateTable
Raw log key 4: [] TCP Job name DeleteData
Initial Group1
Raw log key 5: Image file of size loaded in seconds.
Raw log key 6: Image file of size saved in seconds.
Raw log key 7: Edits file of size edits # loaded in seconds.
Initial Group2
Splitting groups
Raw log key 1: [] TCP Job name UpdateIndex
Raw log key 2: [] TCP Job name DropTable
Raw log key 3: [] TCP Job name UpdateTable
Raw log key 4: [] TCP Job name DeleteData
Raw log key 5: Image file of size loaded in seconds.
Raw log key 6: Image file of size saved in seconds.
Raw log key 7: Edits file of size edits # loaded in seconds.
Extracting log keys
log key 1: [] TCP Job name
log key 2: Image file of size loaded in seconds.
log key 3: Image file of size saved in seconds.

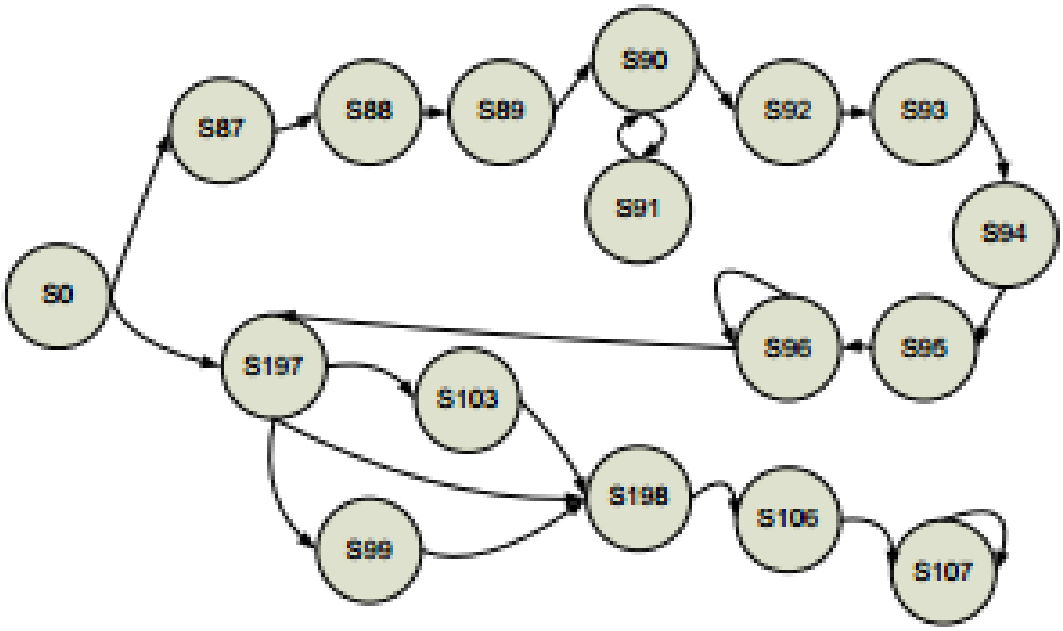


Figure 4.3: Examples of Finite State Machine

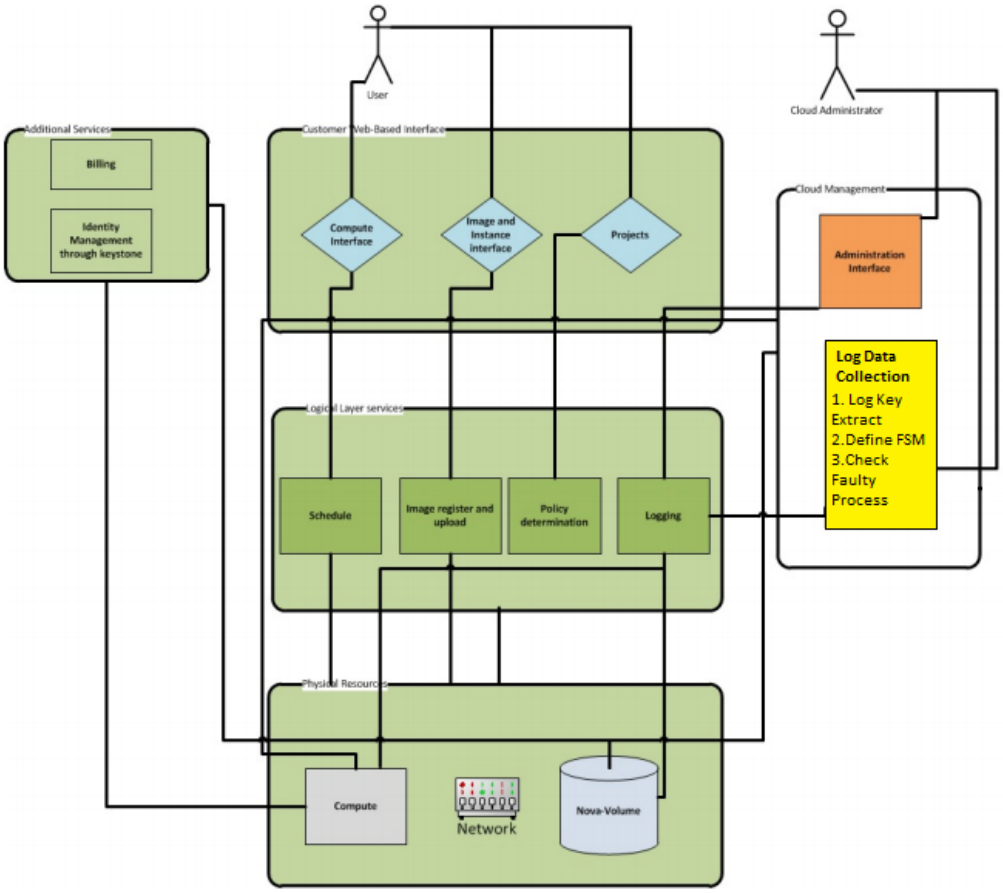


Figure 4.4: Extension of OpenStack Model

Chapter 5

Implementation and Result Analysis

The chapter provides the platform and experimental environment to execute complex re-search experiments and analyze the performance of those. The testbed principally allows implementation of the research and evaluation of the experimental results.

The testbed for the research under consideration involves physical servers of considerable capacity. The cloud environment setup in those consists of OpenStack cloud service which is open source. The latest version of OpenStack Essex release has been used in the latest Ubuntu 12.04 Long Term Service (LTS) servers for the experimental excellence. The cloud controller or compute service is the main process that controls launching and termination of virtual machine instances in OpenStack.

OpenStack Compute which is also called the cloud controller consists of several main components. A cloud controller contains many of these components, and it represents the global state and interacts with all other components. An API Server acts as the web services front end for the cloud controller [33]. The compute controller provides compute server resources and typically contains the compute service. The object store component optionally provides storage services. A volume controller provides fast and permanent block-level storage for the compute servers. Anetwork controller provides virtual networks to enable compute servers to interact with each other and with the public network. A scheduler selects the most suitable compute controller to host an instance. The remaining sections of this chapter provide description of the above components to a greater detail.

5.1 Cloud Environment: OpenStack

OpenStack is a collection of open source technology that provides massively scalable open source cloud computing software. Currently OpenStack develops two related projects: OpenStack Compute, which offers computing power through virtual machine and network management, and OpenStack Object Storage which is software for redundant, scalable object storage capacity. Closely related to the OpenStack Compute project is the Image Service project, named Glance. OpenStack can be used by corporations, service providers, VARS, SMBs, researchers, and global data centers looking to deploy large-scale cloud deployments for private or public clouds [53].OpenStack offers open source software to build public and private clouds. OpenStack is a community and a project as well as open source software to help organizations run clouds for virtual computing or storage [34].

OpenStack Compute is built on a shared-nothing, messaging-based architecture. You can run all of the major components on multiple servers including a compute controller, volume controller, network controller, and object store or image service. A cloud con-troller communicates with the internal object store via HTTP (Hyper Text Transfer Protocol), but it communicates with a scheduler, network controller, and volume controller via AMQP (Advanced Message Queue Protocol) which is also called RabbitMQ server. To avoid blocking each component while waiting for a response, OpenStack Compute uses asynchronous calls, with a call back that gets triggered when a response is received. To achieve the shared-nothing property with multiple copies of the same component, Open-Stack Compute keeps all the cloud system state in a database.

There are currently three main components of OpenStack: Compute, Object Storage, and Image Service. OpenStack Compute is a cloud fabric controller, used to start up virtual instances for either a user or a group. It is also used to configure

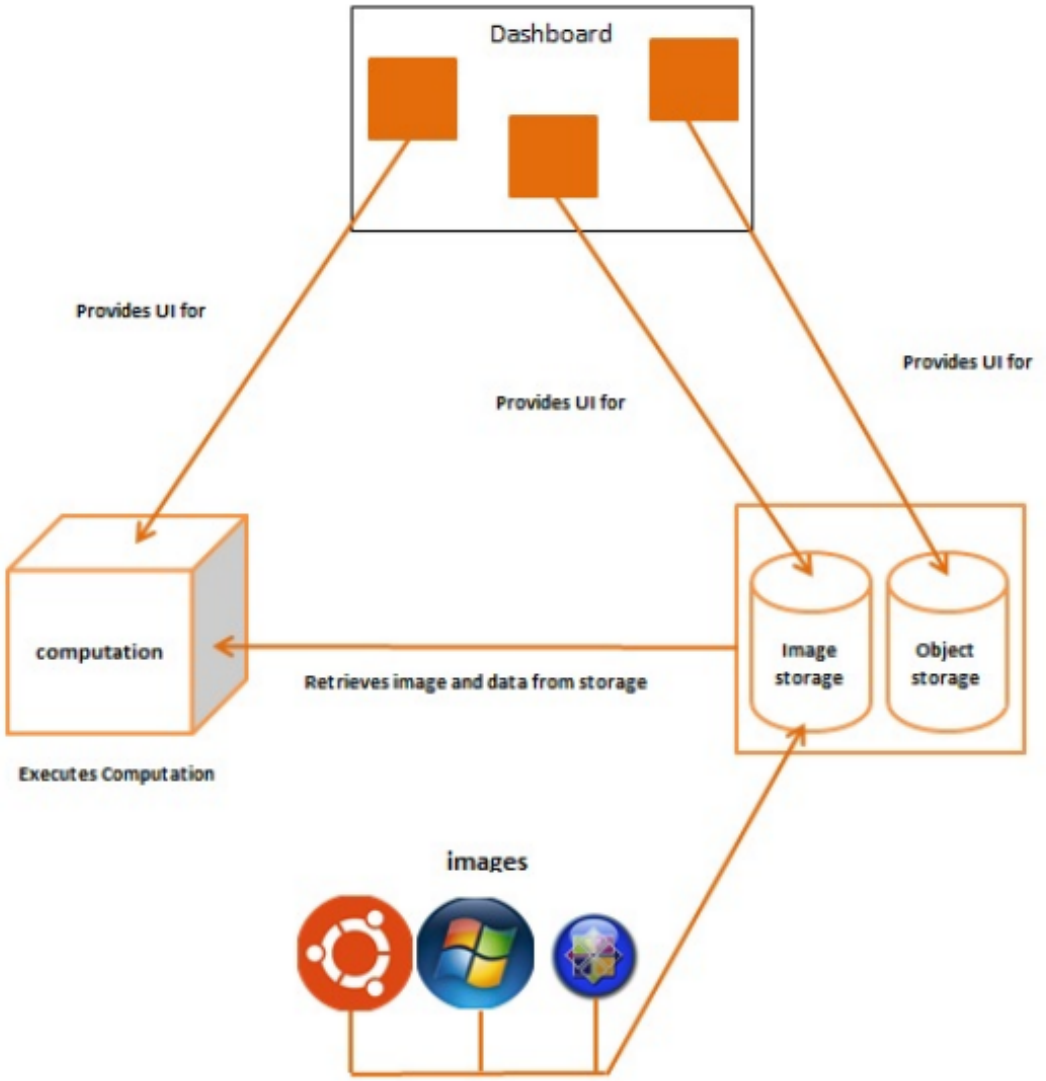


Figure 5.1: Communication between the various components of the cloud infrastructure used for the research

networking for each instance or project that contains multiple instances for a particular project.

OpenStack Object Storage is a system to store objects in a massively scalable large capacity system with built-in redundancy and failover. Object Storage has a variety of applications, such as backing up or archiving data, serving graphics or videos, storing secondary or tertiary static data, developing new applications with data storage integration, storing data when predicting storage capacity is difficult, and creating the elasticity and flexibility of cloud-based storage for your

web applications [11].

OpenStack Image Service is a lookup and retrieval system for virtual machine images. It can be configured in three ways: using OpenStack Object Store to store images; using Amazons Simple Storage Solution (S3) storage directly; or using S3 storage with Object Store as the intermediate for S3 access. Figure 5.1 below shows the communication between the principal components of OpenStack.

As seen in the above diagram, the services of OpenStack cloud can be divided into two major components: Computation which involves providing virtual machines instances to the users in which computational operations are executed and results are obtained, and Storage which provides repository for cloud images and also customer data. The two services are described in the following to a greater detail.

5.2 OpenStack Compute

As stated above, the nova-compute process is primarily a worker daemon that creates and terminates virtual machine instances via Application Interfaces (APIs) of the hypervisor such as XenAPI for XenServer/XCP, libvirt for KVM or QEMU, and VMwareAPI for VMware. The nova-compute consists of a number of other processes like nova-volume, nova-network, nova-schedule, and a Database Management System (DBMS). The process by which this is done is fairly complex and the basics are described below.

1. The compute aims to accept actions from the queue and then perform a series of system commands like launching a Kernel Virtual Machine (KVM) instance while updating state in the database.

2. The nova-volume process manages the creation, attaching and detaching of persistent volumes to compute instance which is similar to the functionality

of Amazons Elastic Block Storage (EBS).

3. The nova-network worker daemon is very similar to nova-compute and nova-volume with primary responsibility of accepting networking tasks from the queue and then performing those to manipulate the network.

4. The nova-schedule process takes a virtual machine instance request from the queue and determines where it should run, more specifically, in which compute server host it should run on.

The SQL database stores most of the build-time and run-time state for a cloud infras-tructure. This includes the instance types that are available for use, instances in use, net-works available and projects. OpenStack cloud infrastructure can support any database supported by SQL-Alchemy but the only databases currently being widely used are sqlite3 (only appropriate for test and development work), MySQL and PostgreSQL. For the re-search on provenance detection, MySQL database is used.

5.3 Storage

The storage infrastructure of the cloud used for the experiment is called swift architecture which is distributed in nature with an aim to prevent any single point of failure as well as to scale horizontally as described below.
Proxy server accepts incoming requests via the OpenStack Object API or just raw HTTP. The proxy server may utilize an optional cache which is usually deployed with memcache to improve performance.
The swift service accepts files to upload, modifications to metadata or container creation. In addition, it will also serve files or container listing to web browsers. The swift service is responsible for maintenance of the account management, con-

Server Processes	Hardware
Cloud Controller node which runs the following: **Network** **Volume** **API**: Provides user interface for enhanced cloud maintenance **Scheduler**: Determines job times and assignment of tasks to processes **Image service**	Following hardware requirement are necessary: **Processor**: 64-bit x86 **Memory**: 16 GB RAM **Disk space**: 1900 GB (SATA or SAS or SSD) **Volume storage**: 100 GB (SATA) for volumes attached to the compute nodes. **Network**: one 1 GB Network Interface Card (NIC) minimum

Figure 5.2: Hardware configuration requirement for the research

tainer management and object management which are responsible for authentication and folder maintenance on the cloud.

With respect to the services mentioned above, account servers manage accounts defined with the object storage service. Container servers manage a mapping of containers within the object store service. Object servers manage actual objects on the storage nodes. There are also a number of periodic processes which run to perform housekeeping tasks on the large data store. The most important of those is the replication services, which ensures consistency and availability through the cluster. Other periodic processes include auditors, updaters and reapers of the database.

The hardware and software environment of the research is important to this respect. The following sections provides a description for those.

5.4 Hardware and Software requirements

The research environment requires a number of hardware and software specifications, the details of which can be described as follows.

Figure 5.3: Nova.conf file of the target research

1. Hardware Specification: The hardware specification is described in the Figure 5.2

2. Software Configuration in the case of Post Installation: Configuring the compute installation involves many configuration files namely the nova.conf file, the api-paste.ini file, and related Image and Identity management configuration files. This section contains the basics for a simple multi-node installation, but Compute can be configured many ways. Networking options and hypervisor options can be described.

3. Setting Configuration Options in the nova.conf file: The configuration file nova.conf is installed in /etc/nova by default. A default set of options are already configured in nova.conf when installed manually. Figure 5.3 identifies the configuration file components.

4. Setting up OpenStack Compute Environment on the Compute Node: *sudo nova-manage db sync* is the command that was used to ensure the database

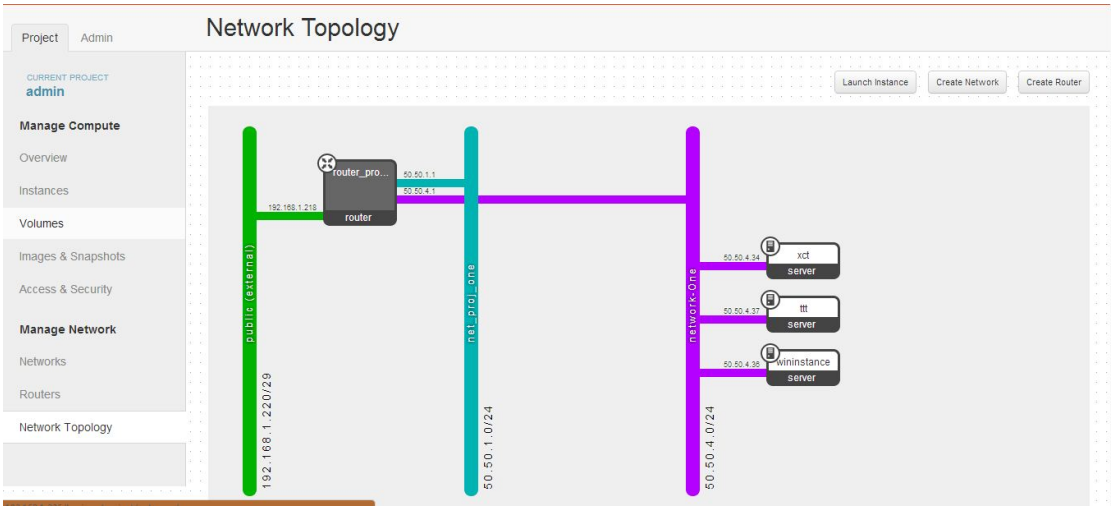


Figure 5.4: Network Structure used for the target research

schema adheres to all the current updates.

5. Software Configuration in the case of Post Installation: Configuring the compute installation involves many configuration files namely the nova.conf file, the api-paste.ini file, and related Image and Identity management configuration files. This section contains the basics for a simple multi-node installation, but Compute can be configured many ways. Networking options and hypervisor options can be described. The requirement exists to populate the database with the network configuration information that Compute obtains from the nova.conf file. Figure 5.4 shows the network structure of the target research.

5.5 Understanding the Compute Service Architecture

The following are some basic categories that describe the service architecture and activities within the cloud controller prepared for the test environment.

1. API Server: At the heart of the cloud framework is an API Server. That

is API Server takes command and control of the hypervisor, storage, and networking programmatically available to users in realization of the definition of cloud computing. The API endpoints are basic http web services which handle authentication, authorization, and basic command and control functions using various API interfaces under the Amazon, Rackspace, and related models. This enables API compatibility with multiple existing tool sets created for interaction with offerings from other vendors. This broad compatibility prevents vendor lock-in hence it was used for the target research.

2. Message Queue: A messaging queue brokers the interaction between compute nodes that are responsible for processing, volumes which are responsible for the block stor-age, the networking controllers which is the software that controls network infras-tructure, API endpoints, the scheduler that determines which physical hardware to allocate to a virtual resource, and similar components. Communication to and from the cloud controller is by HTTP requests through multiple API endpoints. A typical message passing event begins with the API server receiving a request from a user. The API server authenticates the user and ensures that the user is permitted to issue the subject command. Availability of objects implicated in the request is evaluated and, if available, the request is routed to the queuing engine for the relevant workers. Workers continually listen to the queue based on their role, and occasionally their type hostname. When such listening produces a work request, the worker takes assignment of the task and begins its execution. Upon completion, a response is dispatched to the queue which is received by the API server and relayed to the originating user. Next the database entries are queried, added, or removed as necessary throughout the process.

3. ComputeWorker: Compute workers manage computing instances on host

machines. Through the API, commands are dispatched to compute workers to execute the following actions in the research environment.

- Running instances
- Terminating instances
- Rebooting instances
- Attaching volumes
- Detaching volumes
- Get console output

5.6 Network Controller

The Network Controller manages the networking resources on host machines. The API server dispatches commands through themessage queue, which are subsequently processed by Network Controllers. Specific operations include the following for the research environment.

- Allocate fixed IP addresses
- Configuring VLANs for projects
- Configuring networks for compute nodes

5.7 Implementation of ByzantineWarrior for Byzantine Fault Detection

In order to implement ByzantineWarrior a prior knowledge of execution path of all related processes were required. The implementation started by gathering console logs of Openstack modules. Total 455MB of logs were collected. Total number of

```
2013-05-27 21:06:56 WARNING nova.db.sqlalchemy.session [req-10424252-d280-4094-9dc6-2238cd60cab0 None None] SQL connection failed. 9 attempts left.
2013-05-27 21:07:06 WARNING nova.db.sqlalchemy.session [req-10424252-d280-4094-9dc6-2238cd60cab0 None None] SQL connection failed. 8 attempts left.
2013-05-27 21:07:16 WARNING nova.db.sqlalchemy.session [req-10424252-d280-4094-9dc6-2238cd60cab0 None None] SQL connection failed. 7 attempts left.
2013-05-27 21:07:27 WARNING nova.db.sqlalchemy.session [req-10424252-d280-4094-9dc6-2238cd60cab0 None None] SQL connection failed. 6 attempts left.
2013-05-27 21:07:37 WARNING nova.db.sqlalchemy.session [req-10424252-d280-4094-9dc6-2238cd60cab0 None None] SQL connection failed. 5 attempts left.
2013-05-27 21:07:47 WARNING nova.db.sqlalchemy.session [req-10424252-d280-4094-9dc6-2238cd60cab0 None None] SQL connection failed. 4 attempts left.
```

Figure 5.5: Warning Message in the nova-volume log

```
2013-05-27 21:08:27 TRACE nova Traceback (most recent call last):
2013-05-27 21:08:27 TRACE nova   File "/usr/bin/nova-compute", line 49, in <module>
2013-05-27 21:08:27 TRACE nova     service.wait()
2013-05-27 21:08:27 TRACE nova   File "/usr/lib/python2.7/dist-packages/nova/service.py", line 413, in wait
2013-05-27 21:08:27 TRACE nova     _launcher.wait()
2013-05-27 21:08:27 TRACE nova   File "/usr/lib/python2.7/dist-packages/nova/service.py", line 131, in wait
2013-05-27 21:08:27 TRACE nova     service.wait()
2013-05-27 21:08:27 TRACE nova   File "/usr/lib/python2.7/dist-packages/eventlet/greenthread.py", line 166, in wait
2013-05-27 21:08:27 TRACE nova     return self._exit_event.wait()
```

Figure 5.6: Wait Message in the nova-compute log

log lines were 121,101,278. This huge amount of of log lines were than converted to log keys. FSM were created from these log keys. Then for similar processes, new logs were tested.

5.7.1 Log Collection

Console logs are print statements mainly written by the developer for debugging. This logs can be a good source of detecting system behavior. For the target research, console logs of Openstack in the test environment were collected for 6 months. A sample WARNING log message is shown in the Figure 5.5. If checked manually, it can be deducted that, warning log in one module effects another module's log message. Figure 5.6shows the wait message in the nova-compute log caused by warning in nova-volume. For persistency, these logs were collected separately in storage.

5.7.2 Log Parsing

Log keys were generated form these log files and a log lines vs. log key graph were generated. From 121,101,278 log lines, 4,000,000 log lines were selected randomly. This was done to test the accuracy of the system. Then the graph were compared

to the graph generated in [4].

Figure 5.7 shows the log key generation characteristics of Qiang FU et al's work. Similarly, the method was implemented for Openstack for this research. The result showed in Figure 5.8. It is identified that there is a sharp pick in both graph. This is due to the large amount of map-reduce work in hadoop and scheduling work of nova-scheduler in Openstack. The similarity of the graph identifies the implementation of the method to be correct.

5.7.3 FSM Creation

A finite state machine was modeled from this log keys. The details of the FSM creation is stated in the ??.

5.7.4 Testing process execution against Standard FSM

Last of all, the new logs were tested against the standard FSM. To do so, log key and FSM of new logs were created and cross matched against the standard FSM. Finally, ByzantineWarrior algorithm were used to detect the Byzantine Fault in Cloud. The result of the test is given in Table5.1. The details of the result is described in the next section.

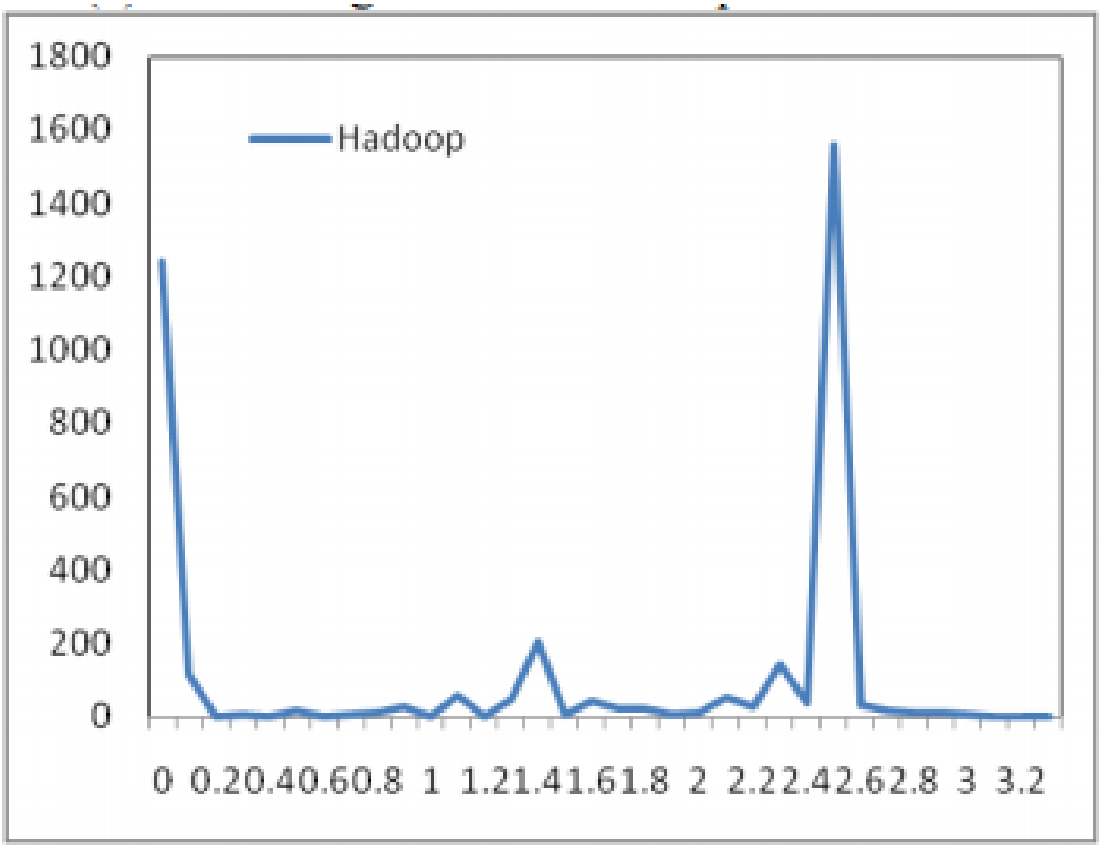


Figure 5.7: Log Key Generation Characteristics of Hadoop's Log

5.8 Precision and Overhead Measurement for ByzantineWarrior

Table 5.1 identifies the compiled results of the experiments. The data obtained are used to identify the precision of the proposed scheme which is determined by finding the Rate of Detection (ROD) of the Fault detection model. For the Total Amount of tests for module x, ϵ_x, the σ_x is determined in terms of the Times Detected variable, τ_x.

$$\sigma_x = \frac{(\tau_x)(100)}{\epsilon_x}, \forall_x \varepsilon X$$

Overhead for any precision, Op of ByzantineWarrior can be used to determine if the obtained precision is favorable in terms of the baseline values, Overhead, Ω_x

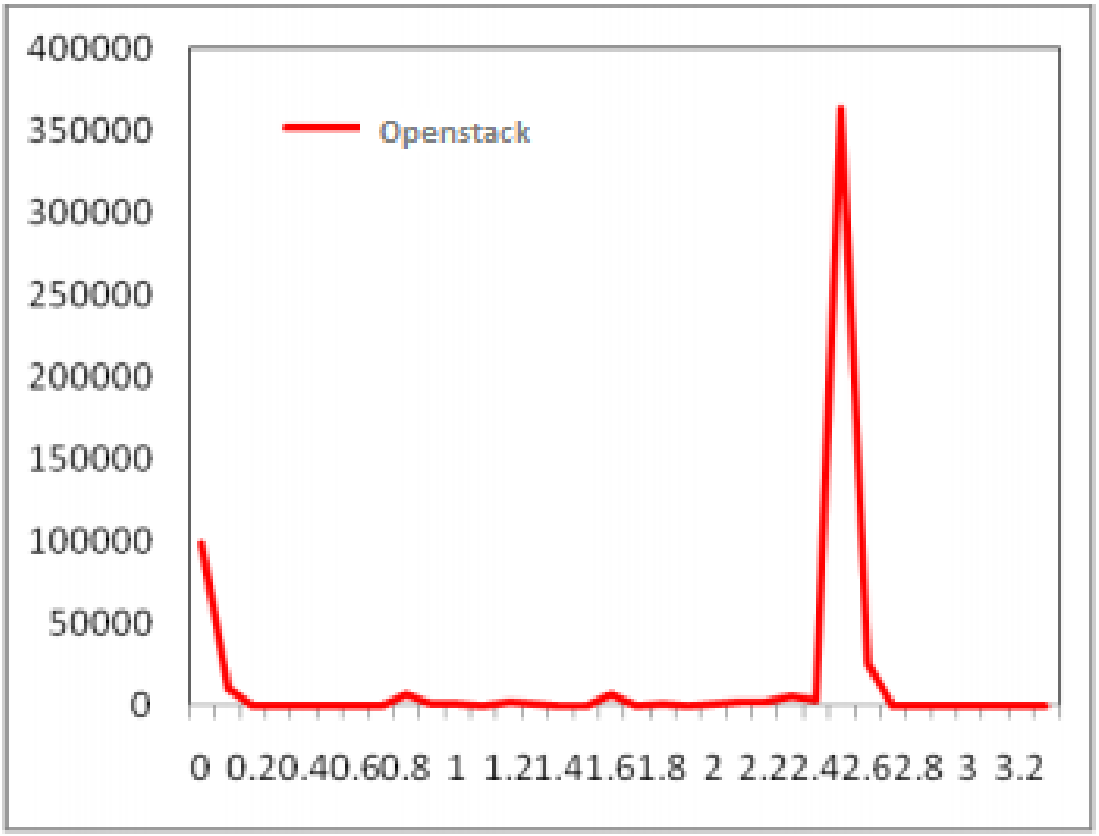


Figure 5.8: Log Key Generation Characteristics of Openstack's Log

for σ_x and Base Line value of x, λ_x is determined by the following formula.

$$\Omega_x = \lambda_x - \sigma_x, \forall_x \varepsilon X$$

The results of this experiment which is tabulated in Table 5.1 satisfies for favorable condition f(x)as shown in the following equation, with the exception of neutron.

$$f(n) = \begin{cases} +favorable & \text{if } \Omega_x \text{is less than or equal 0} \\ -favorable & \text{if } \Omega_x \text{ is greater than 0} \end{cases}$$

Based on the table 5.1, the precision graphs of ByzantineWarrior for the 4 modules for this experiment is shown in Figure 5.9 . The precision Result shows that module is capable of detecting Byzantine Failure with 80 percent accuracy with exception

Module Name	Type	Round	Detected	Precision	Baseline[]	Overhead
nova-compute	compute	200	174	87	86	-.01
nova-scheduler	Scheduling	200	183	91.5	85	-.65
glance-api	image repo	0	0	0	na	0
neutron	network	200	187	93.6	93.8	.002

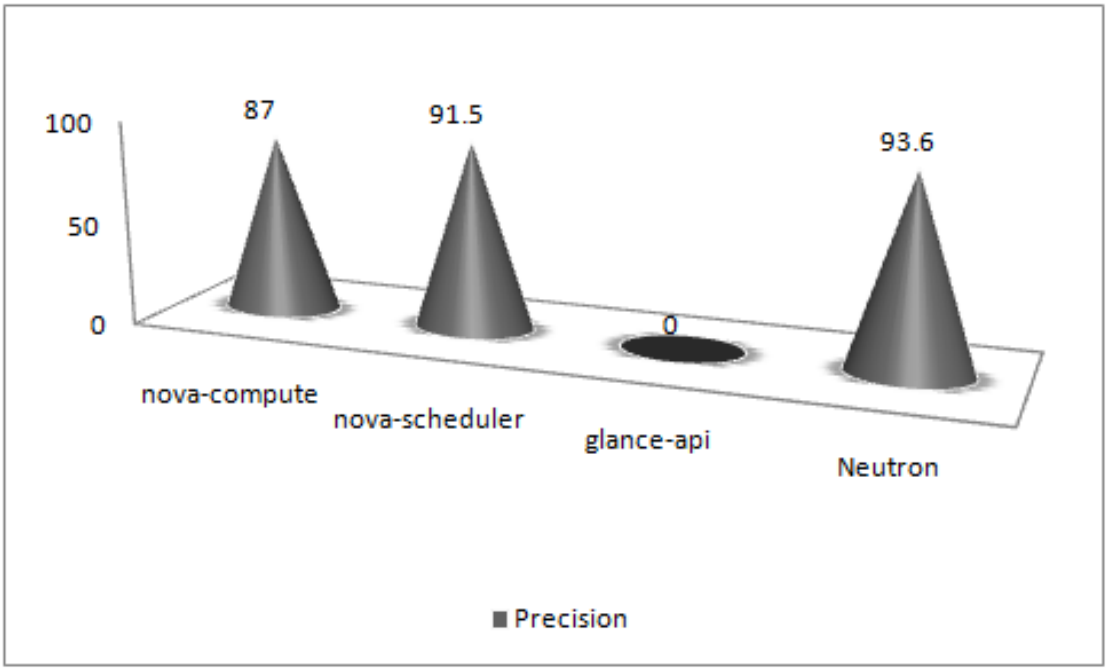


Figure 5.9: Precision Graph of ByzantineWarrior

of glance. The reason for exception is glance only deals with image repository. In this test not so much virtual machines were created. So, the number of log lines were not enough. Also there was no source of baseline value for glance. The baseline values are taken from the research work of []. Figure 5.10 shows the comparative study of precision and base line values.

It is clear from table 5.1 that ByzantineWarrior can detect arbitrary failure in Openstack with over 80 percent accuracy in case of nova-compute and nova-scheduler. It is also clear from the figure 5.9 that ByzantineWarrior gives best result for nova-scheduler.

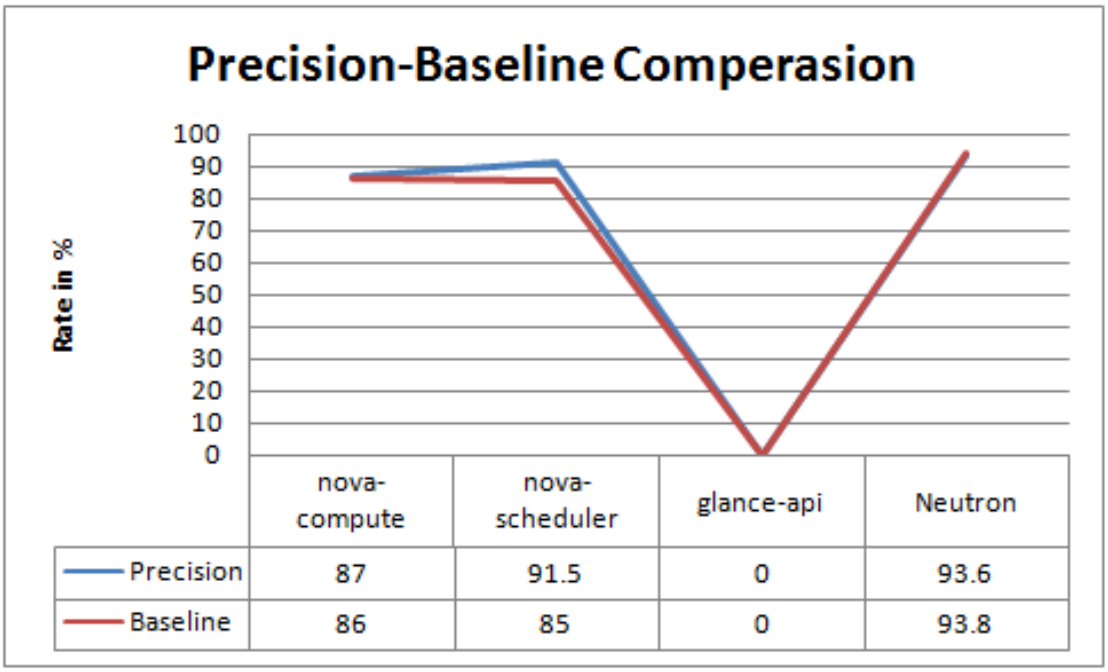


Figure 5.10: Comparison of Precision of ByzantineWarrior against the Base line value Refereed in[]

Chapter 6

Conclusion and Future Work

The research aimed to ascertain a solution to the bottleneck of detecting Byzantine Failure in a widely decentralized system such as cloud computing. The research succeeded in devising ByzantineWarrior, which is a blueprint to the solution of the stated problem and provides a console log based arbitrary detection technique for open source cloud computing.

6.1 Discussion

In the experimentation, ByzantineWarrior successfully analyzed console log files of cloud controllers and nodes. The detected data was used to render useful fault detection information that enable cloud administrators detect arbitrary failure in open source cloud more effectively. As a result, the research show that ByzantineWarrior enable cloud administrators manage the open source cloud infrastructure more effectively.

Empirical investigation was carried out to evaluate the performance of ByzantineWarrior in open source cloud environments, in which a precision rate of 93.8 percent was obtained for detecting neutron-network and 87 percent for nova-compute. Upon comparison of obtained test results with pre-defined baseline and benchmark values desirable overheads of -0.002 and -0.01 were obtained for neutron and nova-compute respectively.This shows that from the performance perspective, ByzantineWarrior has acceptable precision rate to be implemented in a commercial cloud infrastructure based on open source cloud.

Compared to traditional fault detection techniques which function in stand alone systems, ByzantineWarrior is capable of detecting Byzantine Fault in a highly vi-

sualized environment like the cloud, which is a strict requirement [79]. However, the results of ByzantineWarrior were obtained through analysis of 4 Openstack modules with their limited amount of execution data. Hence the research was result-oriented and aimed to provide a generalized solution of Byzantine fault detection for open source OpenStack cloud.

6.2 Future Work

As stated earlier, ByzantineWarrior detects arbitrary failure of Openstack's hand full of modules. More research should be initiated for capturing arbitrary fault for other modules like volume and horizon. Besides, a visual tool should be developed for better view of the detection.

ByzantineWarrior only deals with detection method.However, this research did not intend to find the solution. The detection method used here is the traditional method of Byzantine Fault detection. The work of [] should be incorporated to implement Practical Byzantine Fault detection algorithm for better result.

Bibilography

[1] J. F. . B. Davidson, "Provenance and scientific workflows: Challenges and opportunities," *Proceedings of ACM SIGMOD*, pp. 1–4, 2008.

[2] R. G. Tan, Soila Kavulya and P. Narasimhan, "Visual, log-based causal tracing for performance debugging of map reduce systems," in *IEEE 30th International Conference on Distributed Computing Systems (ICDCS),*, pp. 424–427, IEEE, 2010.

[3] 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.

[4] Y. W. J. L. Qiang FU, Jian-Guang LOU, "Execution anomaly detection in distributed systems through unstructured log analysis," *Ninth IEEE International Conference on Data Mining*, 2009.

[5] L. Riungu, O. Taipale, and K. Smolander, "Practical byzantine fault tolerance and proactive recovery," in *ACM Transactions on Computer Systems*, pp. 398–461, ACM, 2002.

[6] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," . *Communications of the ACM*, vol. 53, 2010.

[7] G. C. V. Christoforos N. Hadjicostis, "Coding approaches to fault tolerance in linear dynamic systems ieee transactions on information theory," *IEEE transactions on information theory*, vol. 51, no. 1, pp. 4–8, 2009.

[8] R. Buyya, C. Yeo, and S. Venugopal, "A time-aware fault tolerance scheme to improve reliability of multilevel phase-change memory in the presence of significant resistance drift," in *IEEE transactions on very large scale integration (vlsi) systems*, pp. 5–13, IEEE, 2008.

[9] R.-T. Wang, "A dependent model for fault tolerant software systems during debugging," . *ieee transactions on reliability*, vol. 61, no. 2, pp. 32–42, 2011.

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

[11] P. Mell and T. Grance, "The nist definition of cloud computing (draft)," *NIST special publication*, vol. 800, p. 145, 2011.

[12] 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.

[13] H. Schaffer, "X as a service, cloud computing, and the need for good judgment," *IT professional*, vol. 11, no. 5, pp. 4–5, 2009.

[14] L. Yu, W. Tsai, X. Chen, L. Liu, Y. Zhao, L. Tang, and W. Zhao, "Testing as a service over cloud," in *Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on*, pp. 181–188, IEEE, 2010.

[15] J. Wu, C. Wang, Y. Liu, and L. Zhang, "Agarica hybrid cloud based testing platform," in *Cloud and Service Computing (CSC), 2011 International Conference on*, pp. 87–94, IEEE, 2011.

[16] L. Zhang, Y. Chen, F. Tang, and X. Ao, "Design and implementation of cloud-based performance testing system for web services," in *Communications and Networking in China (CHINACOM), 2011 6th International ICST Conference on*, pp. 875–880, IEEE, 2011.

[17] T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, T. Hanawa, and M. Sato, "D-cloud: Design of a software testing environment for reliable distributed systems using cloud computing technology," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pp. 631–636, IEEE Computer Society, 2010.

[18] 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.

[19] 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.

[20] J. Sauro and E. Kindlund, "A method to standardize usability metrics into a single score," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 401–409, ACM, 2005.

[21] "Cloud computing software from eucalyptus — leader in cloud software." `http://www.eucalyptus.com`. Accessed: 11 August 2012.

[22] "Digg - what the internet is talking about right now." `http://digg.com`. Accessed: 12 September 2012.

[23] L. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 50–55, 2008.

[24] V. Choudhary, "Software as a service: Implications for investment in software development," in *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pp. 209a–209a, IEEE, 2007.

[25] "Cloud computing: defined and demystified," *IBM Global Services*, 2008.

[26] "Amazon elastic compute cloud (amazon ec2)." `http://aws.amazon.com/ec2`. Accessed: 8 August 2012.

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

[28] "Amazon mechanical turk - welcome." `https://www.mturk.com/mturk/welcome`. Accessed: 15 September 2012.

[29] S. Bhardwaj, L. Jain, and S. Jain, "Cloud computing: A study of infrastructure as a service (iaas)," *International Journal of engineering and information Technology*, vol. 2, no. 1, pp. 60–63, 2010.

[30] 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.

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

[32] 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.

[33] "QEMU." `http://wiki.qemu.org/Main_Page`. Accessed: 13 September 2012.

[34] 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*, pp. 718–723, IEEE, 2012.