Phoenix: International Multidisciplinary Research Journal
Vol 1, No. 3, July-Sep, 2023
ISSN:2583-6897

CLOUD RAID DETECTING DISTRIBUTED CONCURRENCY BUGS VIA LOG MINING AND ENHANCEMENT

First Author[1], Second Author[2]

[1] *Arati, Master of Computer Application BKIT-Bhalki*
[2]Prof .Poojarani, Master of Computer Application BKIT-Bhalki

Abstract -Cloud systems are plagued with distributed concurrency problems, which frequently result in data loss and service outages. CLOUD-RAID, a novel automated tool for discovering distributed concurrency problems fast and effectively, is presented in this work. Distributed concurrency problems are notoriously difficult to detect because they are caused by unexpected message orderings across nodes. CLOUDRAID analyzes and tests automatically just the message orderings that are likely to disclose flaws in cloud systems to discover concurrency bugs in cloud systems fast and effectively. CLOUDRAID specifically mines logs from past runs to identify message orderings that are possible but have not been thoroughly tested. In addition, we offer a log augmenting approach for automatically introducing additional logs into the system under test. These additional logs boost CLOUDRAID's efficacy even further without imposing any apparent performance overhead. Because of our log-based methodology, it is well-suited for live systems. CLOUDRAID was used to investigate six exemplary distributed systems: Three of the nine new problems discovered, all of which have been validated by their original developers, are critical and have already been repaired.

***Key Words*:** Distributed Systems, Concurrency Bugs, Bug Detection, Cloud Computing.

1.INTRODUCTION

Scale-out computing frameworks [1], [2], distributed key-value stores [3], [4], scalable file systems [3], [4], and cluster management services [2] are crucial components of current cloud applications. Because cloud apps provide consumers with online services 24 hours a day, 7 days a week, strong reliability of their underlying distributed systems is critical. Distributed systems, on the other hand, are notoriously difficult to get correctly. There are several software vulnerabilities in real-world distributed systems that frequently cause data loss and cloud outages, costing service providers millions of dollars every incident.

Distributed concurrency issues are among the most troublesome sorts of defects in distributed systems. [7], [8]. These issues are caused by complicated message interleavings, or unexpected orderings of communication events. Concurrent executions on several computers are challenging for programmers to think about and handle effectively. This fact has inspired a huge body of research on distributed system model checkers [9], [10], [11], [12], which uncover difficult-to-find problems by methodically testing all conceivable message orderings. In theory, these model checkers can ensure dependability while performing the same workload that was previously tested. Distributed system model checkers, on the other hand, confront the state-space explosion problem [9]. Despite recent breakthroughs [9], scaling them to many big real-world applications remains problematic. In our trials for executing the WordCount workload on Hadoop2/Yarn, for example, 5,495 messages are involved. Even in such a simple scenario, testing thoroughly all alternative message orderings in a timely manner becomes problematic.

A unique technique for detecting distributed concurrency problems is proposed in this study. Instead of exhaustively testing all potential message orderings, we test only those message orderings that are likely to disclose flaws. Which message orderings are most likely to result in errors? We answer this fundamental question based on two observations:.

2. Literature survey:

MapReduce is a computing model and an application for working with and making big collections. It can be used for a wide range of real-world jobs. Users describe the computation in terms of a map and a reduce function, and the underlying runtime system automatically parallelizes the computation across large clusters of machines, handles machine failures, and schedules communication between machines to make the best use of the network and disks. The system is easy for programmers to use. Over the past four years, Google has built more than 10,000 different MapReduce programs. On average, 100,000 MapReduce jobs are run every day on Google's clusters, processing more than 20 petabytes of data every day.

The initial design of Apache Hadoop [1] was tightly focused on running huge, MapReduce jobs to handle a web crawl. For increasingly diverse companies, Hadoop has become the data and computational agorá---the de facto place where data and computational resources are shared and viewed. This broad adoption and ubiquitous usage has stretched the initial design well beyond its intended target, exposing two key shortcomings: 1) tight coupling of a specific programming model with the resource management infrastructure, forcing developers to abuse the MapReduce programming model, and 2) centralized handling of jobs' control flow, which resulted in endless scalability concerns for the scheduler.

Cassandra is a distributed storage system that offers highly available service with no single point of failure while storing extremely large amounts of structured data dispersed over many commodity machines. Cassandra intends to operate on top of a network made up of numerous nodes (perhaps dispersed across various data centers). Small and large components consistently fail at this scale. The dependability and scalability of the software systems utilizing this service are directly impacted by how Cassandra handles the persistent state in the event of these failures. Although Cassandra shares many design and implementation techniques with databases and resembles them in many respects, it does not enable a full relational data model. Instead, it offers customers a straightforward data model that supports dynamic control over data.

3. Methodology

Ideally, we would like to precisely recover runtime message sequences from existing logs, as annotated in Figure 3. Every logEach instance corresponds to a single static message (or a set of static messages). In this section, log instances from the same run are grouped together.order. In reality, we do source code and log analysis.To retrieve such communication sequences, we must work together. We do statistical analysis.The manner in which static messages are handled and recorded. Instances of runtime logscan then be translated to static messages based on static analysis data. By evaluating logs from the same run, we may group them together.The relationship between recorded variable values and their runtime values based on static dependency analysis. Section 3 contains thetechnical information.The recovered message sequences are then mutated for furthertesting.

Discussion. Our technique just changes the order of two messages. We do not target issues that emerge as a result of several messages being sent in the wrong order. Previous research [30] shown that the majority of distributed concurrency issues are caused by a single out-of-order message, with only 26% requiring more than two messages.

4. SYSTEM ANALYSIS:

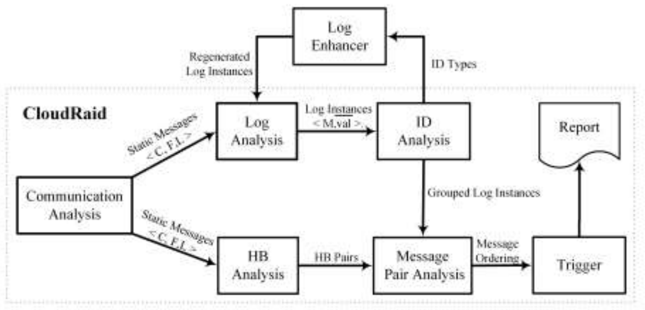
Existing System:

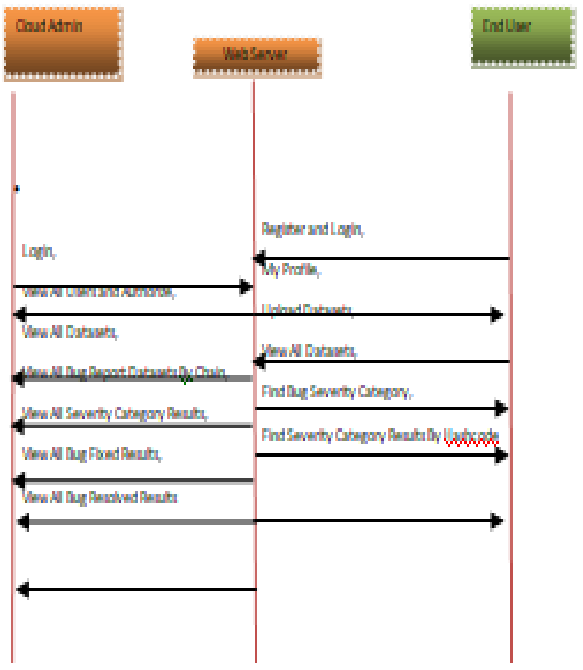
This fact has motivated extensive research on distributed system model checkers that detect elusive faults by systematically exercising all possible message orders. In theory, these model checkers can guarantee reliability when running the same workloads validated previously. However, verifiers of distributed system models face the problem of state-space explosion. Despite recent progress, they remain difficult to scale to many large-scale real-world applications. For example, an experiment running the Word Count workload on Hadoop2/Yarn involved 5,495 messages. Even in such a simple case, it becomes impractical to exhaustively test all possible message orderings in a timely manner.

Proposed System

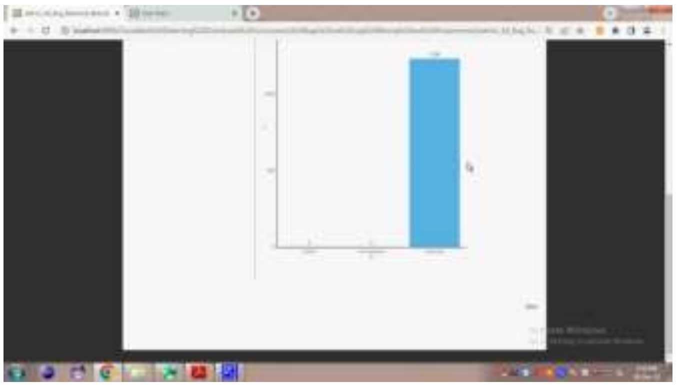
Introducing his CLOUD-RAID, a simple but effective tool for detecting distributed concurrency errors. CLOUDRAID achieves its efficiency and effectiveness by analyzing message sequences that are likely to find errors in existing logs. Our evaluations show that CLOUDRAID is easy to deploy and effective in detecting errors. Specifically, CLOUDRAID tested 60 versions of 6 representative systems in 35 hours and successfully found 31 bugs, including 9 new bugs never reported before.

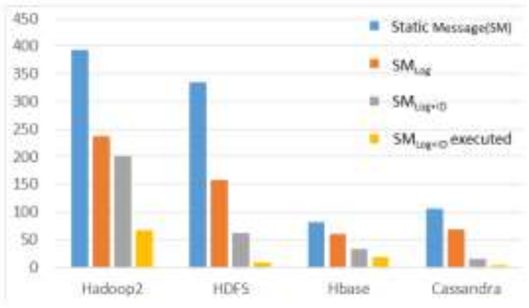
4. ARCHITECTURE




Cloud Admin
Web Server
End User
Register and Login,
Login,
My Profile,
View All Users and Authorize,
Upload Datasets,
View All Datasets,
View All Datasets,
View All Bug Report Datasets By Chain,
Find Bug Severity Category,
View All Severity Category Results,
Find Severity Category Results By Hashcode
View All Bug Fixed Results,
View All Bug Resolved Results

6. Results and Analysis:





Working:

Management reporting in a defect management system is a multi-step process that begins with defect prevention and continues through delivery baseline, defect identification, defect resolution, process improvement, and management reporting.
Optimisation strategies, procedures, and guidelines for fixing flaws in production. The developer's ACKnowledgement of their own responsibility for identifying and fixing problems in deliverables. When an issue is found, it is said to have been "discovered." The purpose of defect resolution process improvement analysis is to reduce the likelihood of similar problems occurring in the future and to communicate defect information to the management system. In order to properly identify issues, the defect prevention process needs to start with a thorough risk assessment.
Organisations should predefine defects by category in the defect discovery process, identify the problem early on before it escalates, and report defects to developers so that they can fix them. Once developers have addressed the problems, the next resolution process will begin, and they will determine the importance of fixing a defect, at which point the schedule process will become very prominent.

Conclusion:

CLOUDRAID, a simple yet effective technique for identifying distributed concurrency issues, is presented. CLOUDRAID's efficiency and efficacy are achieved by assessing message orderings that are likely to disclose faults from existing logs. CLOUDRAID is straightforward to implement and excellent at discovering problems, according to our evaluation. CLOUDRAID, in instance, can test 60 versions of six sample systems in 35 hours, successfully detecting 31 flaws, including 9 previously unknown problems.

In terms of the quantity of gas used to carry out the various operations that are triggered within the smart contract, we have shown that our suggested method is economical. Additionally, the results of the security analysis conducted have demonstrated that our suggested solution achieves protection against malicious attempts targeting the integrity, availability, and nonrepudiation of transaction data, all of which are crucial in a complex multi-party setting like the pharmaceutical supply chain.

In order to achieve end-to-end transparency and verifiability of drug use, we intend to extend the suggested system as part of our ongoing efforts to improve the efficiency of pharmaceutical supply chains.

ACKNOWLEDGEMENT
This work is supported by the National Key research and development program of China (2016YFB1000402), the Innovation Research Group of National Natural Science Foundation of China (61521092 and 61672492), and the National Natural Science Foundation of China (U1736208).

REFERENCES

[1] 2018. Google Protocol Buffer. (2018). Retrieved April 26, 2018 from https: //developers.google.com/protocol-buffers/.

[2] 2018. WALA Home page. (2018). Retrieved April 26, 2018 from http://wala. sourceforge.net/wiki/index.php/Main_Page/.

[3] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, Arvind Krishnamurthy, and Thomas E Anderson. 2012. Mining temporal invariants from partially ordered logs. ACM SIGOPS Operating Systems Review 45, 3 (2012), 39–46.

[4] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. Leveraging existing instrumentation to automatically infer invariantconstrained models. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, 267–277.

[5] Dhruba Borthakur et al. 2008. HDFS architecture guide. Hadoop Apache Project 53 (2008).

[6] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. 2014. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services.. In OSDI. 217–231.

[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[8] Florin Dinu and TS Ng. 2012. Understanding the effects and implications of compute node related failures in hadoop. In Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing. ACM, 187– 198.

[9] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, 1285–1298.

[10] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. 2012. Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12). IEEE Computer Society Press, Los Alamitos, CA, USA, Article 78, 12 pages. http://dl.acm.org/citation.cfm?id=2388996.2389102

[11] Cormac Flanagan and Stephen N Freund. 2009. FastTrack: efficient and precise dynamic race detection. In ACM Sigplan Notices, Vol. 44. ACM, 121–133.

[12] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on. IEEE, 149–158.