A Runtime Verification Based Trace-Oriented Monitoring Framework for Cloud Systems

Jingwen Zhou*†, Zhenbang Chen*†, Ji Wang*†, Zibin Zheng‡§, and Wei Dong†
*Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Changsha, China
†College of Computer, National University of Defense Technology, Changsha, China
‡Shenzhen Research Institute, The Chinese University of Hong Kong, Shenzhen, China
§Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China
Email: {jwzhou, zbchen}@nudt.edu.cn

***Abstract*—Cloud computing provides a new paradigm for resource utilization and sharing. However, the reliability problems, like system failures, often happen in cloud systems and bring enormous loss. Trace-oriented monitoring is an important runtime method to improve the reliability of cloud systems. In this paper, we propose to bring runtime verification into trace-oriented monitoring, to facilitate the specification of monitoring requirements and to improve the efficiency of monitoring cloud systems. Based on a data set collected from a cloud storage system in a real environment, we validate our approach by monitoring the critical properties of the storage system. The preliminary experimental results indicate the promise of our approach.**

***Keywords*-Trace-Oriented Monitoring; Runtime Verification; Failure Detection; Cloud Computing**

I. Introduction

Cloud computing denotes an on-demand way of managing and sharing the system resources of different levels. Nowadays, many cloud services are provided in public. However, the reliability problem of cloud systems challenges the success of cloud computing. For example, in August 2013, the systems of Amazon, Apple, Google and Microsoft crashed successively, in which Google lost $550,000$ dollars in 5 minutes, while Amazon lost 7 million dollars in 100 minutes [1]. Actually, cloud systems are usually large-scale distributed systems, which naturally need runtime methods to improve the reliability besides the methods in design phases. User request trace-oriented monitoring [2] is an important one of such runtime methods, in which traces log the information of handling user requests crossing different nodes of a cloud system, such as function invocations and communications between nodes. Based on traces, many activities can be carried out, including system understanding, failure detection, fault diagnosis, system recovery, *etc*.

Currently, many trace-oriented monitoring frameworks, such as Dapper [3] and Zipkin [4], are used to monitor real world large-scale cloud systems. As we understand, there are following two aspects that need to be further investigated. 1) *The method for specifying monitoring requirements.* The developers and the administrators are pretty familiar with the related cloud systems and can accurately describe the system features. The computers have the ability of exploring latent behaviors of cloud systems. Therefore, the monitoring requirements, both from manual input and machine learning, are very helpful for system monitoring. However, existing specification methods are usually not sufficient to accurately and flexibly express these monitoring requirements. For example, Pip [5] describes the programmer expectations with a declarative language, which is weak in expressing complex features, *e.g.*, temporal requirements. In IRONModel [6], the knowledge about the queuing theory and entropy test is required in specifying monitoring requirements, which is a tough process for general users. 2) *The efficiency of monitoring.* The public service of a cloud system usually receives thousands of user requests in a very short time. A request may be handled in a complex process. For example, a simple Google search request will trigger more than 200 sub-requests and cross hundreds of servers. Therefore, massive trace data would be produced in cloud systems, which is a real problem for real-time monitoring. Actually, the efficiency of monitoring challenges all existing monitoring methods.

To facilitate these issues, we propose a framework that brings runtime verification (RV) into the field of trace-oriented monitoring in cloud systems. The monitoring requirements of cloud systems can be specified by formal specification languages, such as Finite State Machine (FSM), Linear Temporal Logic (LTL) [7], and the temporal logic of Calls and Returns (CaRet) [8]. The monitors for these critical requirement properties can be effectively generated, and then the monitoring of traces can be efficiently carried out. Based on a data set, called TraceBench [9], collected on a real world cloud storage system, *i.e.*, Hadoop Distributed File System (HDFS) [10], we use our framework to monitor different representative kinds of HDFS properties. The preliminary experimental results are promising.

The rest of this paper is structured as follows. In Section II, we introduce trace-oriented monitoring. Section III describes our approach and some preliminary results, and

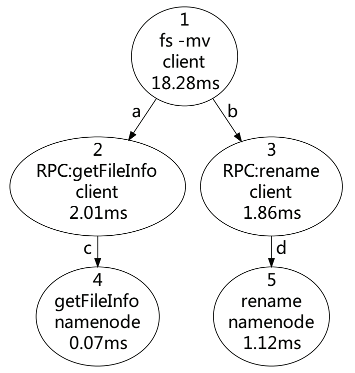

Figure 1. A trace sample [9]

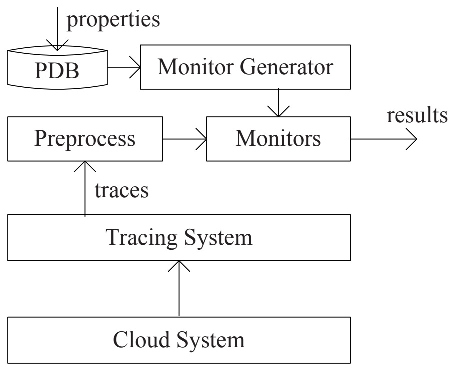

Figure 2. Basic approach

Section IV concludes the paper and presents the future work.

II. Trace-Oriented Monitoring

Trace-oriented monitoring takes user request traces, or simply called traces, as data source. A trace, tracking the execution path of a user request, consists of events and the relationships [11], where an event records the details of one execution step in handling the user request, such as function name, execution latency and result, and a relationship records the causal relation between two events, like local and remote function calls. With the events and the relationships, a trace can be constructed into a trace tree. Fig. 1 is a trace sample in the form of a trace tree, recording the execution process of a move request in HDFS, *i.e.*, *fs -mv*. The process involves two machines: the *client*, who sends the move request to HDFS, and the *namenode*, which is the master server for managing the files in HDFS. The *client* first gets the information of the target file from the *namenode* using a Remote Procedure Call (RPC), *i.e.*, RPC: *getFileInfo*, to check if the file exists, and then rename the file. Five events, corresponding to the five nodes in the tree, record the execution details of five functions. The edges represent the relationships between events. For example, the edge *a* expresses a local function call, since the father node and the child node are produced on a same machine, and the edge *c* represents a remote function call, since node 2 and node 4 belong to different machines. Naturally, a trace can be transferred to a linear event sequence using different methods, such as the Depth First Search (DFS) or using Call (c) and Return (r) to describe an event. For example, the trace in Figure 1 can be expressed as “1,2,4,3,5” using DFS, or “$c_1c_2c_4r_4r_2c_3c_5r_5r_3r_1$” using the call and return method.

The traditional resource-oriented monitoring methods record the resource consumption information of systems, such as memory usage and CPU speed. In contrast, traces record some more find-grained information about system executions, such as remote function calls and execution time. Basing on traces, some more fine-grained monitoring activities can be carried out, including stateful requirement monitoring, performance problem detection and diagnosis, complex feature understanding, *etc*.

III. Using RV in Trace-Oriented Monitoring

In this section, we first introduce our basic approach, and then describe the trace data set supporting this research and some critical properties we found in this data set, which validate our idea. Lastly, we give some experimental results that indicate the promise of our work.

A. Basic Approach

Figure 2 shows the basic approach of our framework. The monitoring requirements of target cloud systems are firstly expressed as properties, using the various specification languages in RV. And the properties are then stored in the property database (PDB). Using the method of generating monitors in RV [12], efficient monitors can be generated for the properties in PDB. On the other hand, when the cloud system runs, traces of system running are collected by certain tracing systems, *e.g.*, X-Trace [13]. Before delivered to the monitors, some preprocess are needed, such as linearizing using DFS and removing irrelevant events. After that, the traces are verified by related monitors to check whether the corresponding properties are violated or satisfied. The monitoring results can then be induced by the checking results, such as system running normally, happening system failures or appearing performance faults. To improve the efficiency and correctness, it also need some specific optimizations, such as sampling, voting and multi-thread, which we leave to our future work.

With a data set of traces collected from HDFS, we conclude some representative critical temporal properties of HDFS, and check the traces in the data set with these properties. The preliminary results indicate that the monitors can efficiently and correctly identify the failed traces.

B. Data Set and Properties

HDFS [10] is a widely used cloud system of file storage. In HDFS, a file is divided into many data blocks that are repetitively stored in different nodes (called datanodes). And there also exists a master node, called namenode, for managing the name space of files. The user requests of HDFS can mainly be classified into three kinds: the *write* request for uploading files to HDFS, the *read* request for downloading files, and the *rpc* request, which only contains the RPC operations (like Figure 1 shows) for file managements, including querying, removing, renaming, *etc*.

In [9], a fine-grained trace data set, called TraceBench, is collected in an HDFS system deployed on a real environment considering different scales, such as cluster size, user request type and speed. To simulate real scenarios, various kinds of faults are also injected during collection, including both function faults and performance faults, such as data block loss and network slowdown. Therefore, the traces in this data set contain the system behaviors of both normally running and abnormally running. TraceBench consists of three classes: the *Normal* class, in which traces are collected when HDFS runs normally; the *Abnormal* class, in which a certain permanent fault is injected during collecting; and the *Combination* class, with some faults being randomly injected and later removed.

Based on our understanding of HDFS, there are many temporal properties for monitoring. Based on TraceBench, we extract tens of such properties. Following are representative samples related to different kinds of requests for detecting system failures or faults.

Property 1. A *rpc* request of HDFS starts with a RPC: *getFileInfo* to get the information of the target file from the namenode, which is followed by some other RPCs for related operations on the file, like *rename* in Figure 1. Therefore, the trace of a file moving request should obey the following LTL property, and a failure occurs when a violation happens.

$$getFileInfo \ \wedge \ \Box(\ getFileInfo \rightarrow \bigcirc(\Diamond\ rename))$$

Where $\Box$, $\bigcirc$ and $\Diamond$ respectively represent for **Always**, **Next** and **Exist** in LTL. In addition, many other *rpc* requests contain similar properties.

Property 2. A *read* request always contains many data block reading operations, each of which starts with calling *blockSeekTo* (denoted by *B* for short). Once a reading operation fails after several retries, the whole request fails. So, for a correctly handled *read* request, the last block should be correctly read, which is indicated by *checksumOK* (*K*). Hence, the following LTL property can express this requirement.

$$\Diamond B \wedge (\Box((B \rightarrow \bigcirc(\Box\neg B)) \rightarrow \bigcirc(\Diamond K)))$$

Where $\Diamond B$ expresses that each *read* request contains at least one reading operation, and $B \rightarrow \bigcirc(\Box\neg B)$ denotes the last reading operation.

Property 3. Similarly, writing a file to HDFS involves multiple data block writing operations, each of which starts

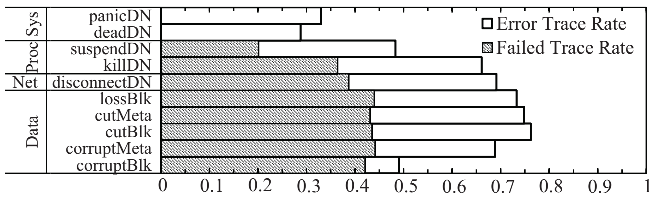

Figure 3. Checking result

by invoking *createBlockOutputStream* (*C*). A normal *write* request should successfully upload the last block, which requires that the number of connected datanodes, indicated by the number of *writeBlock* (*W*), equals to the number of repetitions stored, by counting *receiveBlock* (*R*). Hence, this property is a context-free property, and CaRet [8] can be used to specify it as follows.

$$\Diamond C \wedge (\Box((C \rightarrow \bigcirc(\Box\neg C)) \rightarrow \bigcirc(\Diamond W \wedge \Box(W \rightarrow \bigcirc^a R))))$$

Where $\bigcirc^a$ is the **abstract Next** operator in CaRet, and $W \rightarrow \bigcirc^a R$ expresses the equality.

Besides for detecting failures, properties can also be used for diagnosing various faults, like *Property 4* shows.

Property 4. When reading a data block, HDFS first selects the best datanode for downloading by calling *bestNode* (N). If the selected datanode is invalid, the selecting process repeats and a second best datanode is picked. Therefore, if the *bestNode* are invoked more than once in reading a data block, we consider some problems may happen in the abandoned datanode(s), such as network disconnected, data block missing and process stopped. This requirement can be expressed as following LTL property.

$$B\,\mathcal{R}\,\neg N \wedge \Box(B \rightarrow \bigcirc(\neg B\,\mathcal{U}\,N))$$
$$\wedge\Box(N \rightarrow (\bigcirc(B\,\mathcal{R}\,\neg N) \vee \neg\bigcirc true))$$

Where $\mathcal{R}$ and $\mathcal{U}$ respectively represent for **Release** and **Until** in LTL.

C. Preliminary Experimental Results

We manually generate the monitors in terms of SQL queries of the *Property 1* and *Property 2*, and use the monitors to check the traces in the data set. Figure 3 illustrates the results of checking the traces of *read* requests. The vertical axis shows the checked trace sets, named by the injected faults of different types, *i.e.*, *System* (*Sys*), *Process* (*Proc*), *Network* (*Net*), and *Data* [9]. In Figure 3, the error trace rates are the percentages of the traces containing errors due to the injected faults. And the failed trace rates, achieved by checking traces with *Property 2*, indicate the percentages of unsuccessfully handled requests caused by errors. The number of the failed traces checked by *Property 2* is exactly the same as the actual number in the trace set, showing the accuracy of the property and the correctness of our framework. Surprisingly, we also find several failed traces in

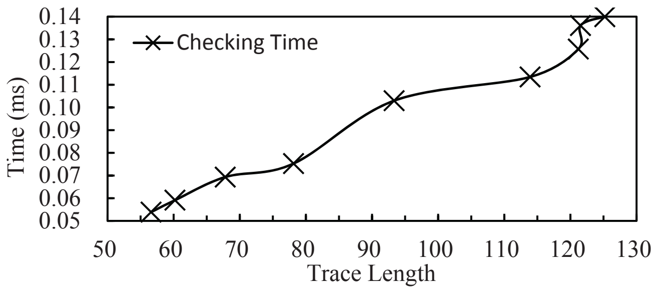

Figure 4. Checking time

the normal trace sets. The reason is that the tracing system drops some events when reaching its handling bottleneck. Therefore, the failed traces are induced by the tracing system errors rather than the cloud system failures. However, it still validates our approach.

We have also evaluated the efficiency of our framework. By checking traces with *Property 2*, Figure 4 shows the relationship between checking time and trace length. For a trace containing 100 events, the checking time is about 0.1 ms, *i.e.*, about $10,000$ traces can be checked in 1 second in this condition. This is a pretty promising result, which can be further improved with various optimizations. Besides the trace length, the efficiency is also related to many other factors, such as the complexity of trace and property, the number of monitors, which should be further investigated.

IV. Conclusion and Future Work

In this paper, we propose a novel approach of introducing runtime verification into the trace-oriented monitoring in cloud systems. Based on TraceBench, we have validated our approach by monitoring some critical properties of HDFS. The experimental results indicate the promise of our approach.

This is an ongoing work. In the future, there are following aspects to work on: first, we will integrate existing RV frameworks into our tracing system, and implement some existing RV methods, such as CaRet RV [14]; second, highly efficient and scalable monitoring algorithms under trace-oriented monitoring scenarios will be investigated; third, since performance problems are important for cloud systems, we will consider to use RV to monitor the performance aspects of cloud systems, where the time information of traces can be used; last, more experiments and more applications on other real world cloud systems are planned.

V. Acknowledgment

This work is supported by the National 973 Program of China under the Grant No.2011CB302603, the National Natural Science Foundation of China (NSFC) under the Grant No. 61161160565 and No. 61103013, and the SRFDP under the Grant No. 20114307120015.

References

[1] J. Garside. Nasdaq crash triggers fear of data meltdown. http://www.theguardian.com/technology/2013/aug/23/nasdaq-crash-data

[2] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, “Pinpoint: Problem determination in large, dynamic internet services,” in *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN 2002)*. IEEE Computer Society, 2002, pp. 595–604.

[3] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, “Dapper, a large-scale distributed systems tracing infrastructure,” Google, Tech. Rep., 2010.

[4] Twitter. Zipkin, from twitter. http://twitter.github.io/zipkin/

[5] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, “Pip: Detecting the unexpected in distributed systems,” in *Proceedings of the 3rd Conference on Networked Systems Design and Implementation (NSDI’06)*. USENIX Association, 2006, pp. 9–9.

[6] E. Thereska and G. R. Ganger, “IRONModel: Robust performance models in the wild,” in *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2008)*. ACM Association, 2008, pp. 253–264.

[7] Z. Manna, *Temporal Verification of Reactive Systems: Safety*, vol. 2. Springer, 1995.

[8] R. Alur, K. Etessami, and P. Madhusudan, “A temporal logic of nested calls and returns,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. Springer, 2004, pp. 467–481.

[9] J. Zhou, Z. Chen, J. Wang, Z. Zheng, and M. R. Lyu, “Towards an open data set for trace-oriented monitoring,” in *Proceedings of the 7th IEEE International Conference on Cloud Computing (CLOUD 2014)*. IEEE Computer Society, 2014, pp. 922–923.

[10] Apache. Hadoop. http://hadoop.apache.org/

[11] J. Zhou, Z. Chen, H. Mi, and J. Wang, “MTracer: A trace-oriented monitoring framework for medium-scale distributed systems,” in *Proceedings of the IEEE 8th International Symposium on Service Oriented System Engineering (SOSE 2014)*. IEEE Computer Society, 2014, pp. 266–271.

[12] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Rosu, “An overview of the MOP runtime verification framework,” *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 249–289, 2012.

[13] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, “X-Trace: A pervasive network tracing framework,” in *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI’07)*. USENIX Association, 2007, pp. 271–284.

[14] N. Decker, M. Leucker, and D. Thoma., “Impartiality and anticipation for monitoring of visibly context-free properties,” in *Proceedings of 4th International Conference on Runtime Verification (RV 2013)*. Springer, 2013, pp. 183–200.