A Data Set for User Request Trace-Oriented Monitoring and its Applications

Jingwen Zhou, Zhenbang Chen, Member, IEEE, Ji Wang, Member, IEEE, Zibin Zheng, Member, IEEE, and Michael R. Lyu, Fellow, IEEE

Abstract—User request trace-oriented monitoring is an effective method to improve the reliability of cloud services. However, there are some difficulties in getting useful traces in practice, which hinder the development of trace-oriented monitoring research. In this paper, we release a fine-grained user request-centric open trace data set, called TraceBench, which is collected in a real-world cloud storage service deployed in a real environment. When collecting, we consider different scenarios, involving multiple scales of clusters, different kinds of user requests, various speeds of workloads, many types of injected faults, etc. To validate the usability and authenticity, we have employed TraceBench in several trace-oriented monitoring topics, such as anomaly detection, performance problem diagnosis, and temporal invariant mining. The results show that TraceBench well supports these research topics. In addition, we have also carried out an extensive data analysis based on TraceBench, which validates the high quality of the data set.

Index Terms—Data set, end-to-end tracing, trace-oriented monitoring, anomaly detection, cloud services

1 INTRODUCTION

Nowadays, cloud systems appear in more and more fields to provide various services that greatly benefit our daily life. However, with the increase in scale and complexity, cloud systems are more and more often experience correctness and performance problems, which affect the reliability of cloud services and may bring enormous loss. For example, in August 2013, the services of Apple, Google, Microsoft, and Amazon crashed for different reasons. Google lost 550,000 dollars in 5 minutes, while Amazon lost seven million dollars in less than 100 minutes [1]. Moreover, some of these problems are very hard to detect in the design stage using traditional methods, such as testing, validation, and verification. Therefore, as a complement, the methods at runtime are used to further improve the reliability of cloud services, where monitoring is a most important one.

The monitoring of a system usually contains the activities of collecting the runtime information of the system, analyzing the system behaviors based on the collected information, and adjusting the system according to the analysis results. Therefore, topics like tracing, anomaly detection, and problem diagnosis are all related to the category of monitoring. According to the collected information, the methods of monitoring can mainly be divided into two types: resource-oriented monitoring and trace-oriented monitoring [2]. Resource-oriented monitoring systems [3], [4] record the information of a system from different kinds of system resources, such as CPU speed and available memory. However, sometimes resource information alone is not sufficient for revealing the behaviors of the monitored system, e.g., to tell how a user request is processed. In contrast, trace-oriented monitoring systems [5], [6], [7] track the processes of handling user requests and record the context of each step, such as function name and execution time, in the form of user requests traces, or simply called traces. Traces naturally record the execution paths of user requests and contain both the normal and abnormal features of the monitored system, which are more valuable for revealing system behaviors.

In recent years, more and more research topics of improving system reliability are based on traces, such as uncovering bugs [8], diagnosing correctness problems [9] and performance problems [10], [11]. However, there still exist some difficulties in getting useful data of traces, which are essential for these research topics. First, collecting traces manually is a tedious and time-consuming process, which may involve choosing or even implementing a collection system, instrumenting and deploying the monitored system, designing and establishing various scenarios, etc. Moreover, these steps may be repeated multiple times if the collected traces are insufficient. For example, the jobs of collecting the trace data in this paper last for more than half a year, showing the difficulties in collecting traces by hand. Second, due to the weakness in authenticity, manually synthesized traces sometimes are not sufficient for the research topics. Finally, in academia and industry, there exist few data sets of traces that can well support these research topics, and companies do not want to release their system traces for safety or other considerations. All the above issues hinder the development of trace-oriented monitoring topics, and also motivate this paper.
In this paper, we release TraceBench to support trace-oriented monitoring research. To the best of our knowledge, TraceBench is the first fine-grained user request-centric open trace data set. The collection system is MTracer, a trace-oriented monitoring system developed by us. With MTracer, we collected TraceBench among the Hadoop Distributed File System (HDFS), a widely used cloud system that provides storage service. The cluster is deployed in a real environment, which is composed by 50 datanodes, 50 clients, and some other hosts. The whole size of TraceBench is about 3.2 GB. It records more than 370,000 traces stored in 364 files. TraceBench consists of three classes. (1) Normal: in this class, the traces are collected when the HDFS runs normally with different cluster sizes and responds to various kinds of user requests in multiple speeds. (2) Abnormal: we inject permanent faults into the HDFS during the collection, which result in correctness and performance problems, such as data block missing and network slowdown. (3) Combination: we randomly inject faults during HDFS running and then perform the recovery, so that the traces record both the normal and abnormal system behaviors.

TraceBench is a well-designed trace data set, considering multiple scales of clusters, different kinds of user requests, various speeds of workloads, many types of injected faults, and so on, and hence can be used in many trace-oriented monitoring topics as Ref. [15] advises, such as anomaly detection, problem diagnosis, and distributed profiling. For example, the Normal class and Abnormal class can be treated as knowledge bases for training the algorithms to learn the features of normal behaviors and abnormal behaviors, respectively, and the Combination class can be employed as a test set for validating the effectiveness of the algorithms. Since TraceBench is collected in a real environment, other research topics, like system understanding, would also employ this data set.

The remainder of this paper is organized as follows. Section 2 introduces trace-oriented monitoring systems and the system used for collecting TraceBench. Section 3 describes TraceBench, focusing on the details of the data set and the details of the collection. In Section 4, we show several applications of TraceBench. Section 5 discusses the aspects that may threaten the validity of TraceBench. Section 6 reviews the related work and Section 7 concludes the paper.

2 TRACING ORIENTED MONITORING

In this section, we first describe the trace-oriented monitoring systems, and then introduce MTracer, the system we used to collect TraceBench.

2.1 Trace-Oriented Monitoring Systems

Trace-oriented monitoring systems can be mainly divided into two types: black-box-based systems and white-box-based systems [2]. The black-box-based systems, such as lprof [16], PreciseTracer [17], and vpath [18], do not require the source code of the monitored system and apply reasoning methods for reconstructing the trace. In contrast, the white-box-based systems, such as Stardust [19], X-Trace [20], and Fay [21], need to instrument the source code at first for recording traces. In practice, the black-box-based systems are less flexible and less accurate than the white-box-based systems.

Different monitoring systems record traces in different formats, such as the linear event sequence, the trace tree [20], and the directed acyclic graph (DAG) [20], [23], [24]. The trace that contains more information often employs a more complex format and brings more overheads in monitoring. For example, among the above three formats, the DAG contains the most information and also is the most complex one. A trace with a complex format can be transformed to a simple format. For example, the trace in the format of trace tree can be converted to the format of linear event sequence using the methods like Depth First Search (DFS).

Among many existing trace-oriented monitoring systems, we choose MTracer to collect TraceBench for the following several reasons. 1) Accuracy. MTracer is a white-box-based system, which can guarantee that the collected traces exactly record the processes of handling user requests. 2) Trace format. MTracer employs the format of trace tree, which balances between recording abundant information and reducing the complexity of the trace format. 3) Usability. MTracer is lightweight for deployment and maintenance, and efficient for monitoring and visualization, which is pretty suitable for our collection.

2.2 MTracer

As Fig. 1 shows, MTracer adopts a client/server architecture, in which the hosts of the monitored system are treated as MTracer clients. Since MTracer is a white-box-based system, we need to instrument the source code of the monitored system at first. When the execution crosses these instrumented codes on the MTracer clients, related information is collected and sent to the MTracer server in the form...
of events and edges, which will be introduced in Section 3. The server extracts useful information from the received data and stores the extracted information with MySQL database [25]. A web-based front-end for visualization is provided, called MTracer-Viz4, to reconstruct the traces, provide various flexible queries, and do some advanced analysis.

By using different mechanisms and optimizations, MTracer requires few additional resources on the clients. For example, MTracer consumes a bandwidth of less than 1 MB/s, which is pretty acceptable comparing to the Gbps-level network in the data center. To improve the efficiency of the server, MTracer adopts a parallel method and several optimizations when storing data. More details about MTracer can be found in Ref. [13].

3 TRACEBench

Using MTracer, we collected TraceBench in an environment containing 50 clients for generating user requests, a Hadoop cluster with 50 datanodes and one namenode for processing the requests from clients, and other hosts. When collecting, five workloads and 17 faults in five types are introduced to collect different behaviors of HDFS. As shown in Table 1, the whole size of TraceBench is about 3.2 GB and the total data collection time is 11,002 minutes. TraceBench consists of 370,334 traces, 14,777,970 events, and 6,303,153 edges in 364 trace files, in which 100 files contain failed user requests. The number of contained events of a trace, or say trace length, in TraceBench spreads from 1 to 420, and the number of involved hosts in a trace spans from 1 to 44. In this section, we first introduce the trace format and the structure of TraceBench, and then give the details about the collection, including the collection environment, the collection process, the employed workloads, and the injected faults.

3.1 Trace Format

In this section, we first introduce the trace tree with a traditional format, and then present the format of the traces in TraceBench, which is a little different from the traditional format. At last, we show some examples in TraceBench.

3.1.1 Trace Tree

Formally, a trace in the form of trace tree can be formalized as \((E, R)\) [2], where \(E\) and \(R\) are the event set and the set of the relationships between the events, respectively. An event records the context of a request step, where a step stands for the execution of a function or a routine. An event is a triple \((tid, eid, I)\). The first element \(tid\) is the identity of the trace, which means that all the events with the same \(tid\) belong to the same trace. Each event has a unique \(eid\) for distinguishing from each other in a trace. \(I\) records the detailed information of the step, such as function name and execution time. A relationship records the causality between two events, and can be expressed in a quadruple \((tid, feid, ceid, T)\), which means the event identified by \(feid\) is the father of the event identified by \(ceid\), in the trace identified

<table>
<thead>
<tr>
<th>Class</th>
<th>Type</th>
<th>Fault</th>
<th>Workload</th>
<th>Variable</th>
<th>#File</th>
<th>Collection Time(min)</th>
<th>#Trace</th>
<th>#Event</th>
<th>#Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal (NM)</td>
<td>Clientload (CL)</td>
<td>-</td>
<td>r/w/r/w</td>
<td>1.5i clients (C)</td>
<td>55</td>
<td>2,761</td>
<td>209,326</td>
<td>3,848,191</td>
<td>1,963,129</td>
</tr>
<tr>
<td></td>
<td>Datanode (DN)</td>
<td>-</td>
<td>r/w/r/w</td>
<td>1.5i datanodes (DN)</td>
<td>33</td>
<td>1,980</td>
<td>17,440</td>
<td>2,310,699</td>
<td>941,681</td>
</tr>
<tr>
<td>Abnormal (AN)</td>
<td>Process (Proc)</td>
<td>killDN</td>
<td>r/w</td>
<td>0,1,2,3,4,5i FDN</td>
<td>30</td>
<td>600</td>
<td>6,469</td>
<td>682,595</td>
<td>240,668</td>
</tr>
<tr>
<td></td>
<td>suspendDN</td>
<td>r/w</td>
<td>1,2,3,4,5i FDN</td>
<td>28</td>
<td>515</td>
<td>2,754</td>
<td>317,490</td>
<td>124,630</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Network (Net)</td>
<td>disconnectDN</td>
<td>r/w</td>
<td>1,2,3,4,5i FDN</td>
<td>28</td>
<td>560</td>
<td>5,189</td>
<td>536,810</td>
<td>192,276</td>
</tr>
<tr>
<td></td>
<td>slowHDFS</td>
<td>r/w</td>
<td>0,10i/2i/100i ms</td>
<td>33</td>
<td>506</td>
<td>41,200</td>
<td>669,602</td>
<td>348,912</td>
<td></td>
</tr>
<tr>
<td></td>
<td>slowDN</td>
<td>r/w</td>
<td>1,2,3,4,5i FDN</td>
<td>28</td>
<td>560</td>
<td>4,395</td>
<td>575,425</td>
<td>232,270</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>corruptBlk</td>
<td>r</td>
<td>0,1,2,3,4,5i FDN</td>
<td>15</td>
<td>300</td>
<td>5,354</td>
<td>636,023</td>
<td>244,296</td>
<td></td>
</tr>
<tr>
<td></td>
<td>corruptMeta</td>
<td>r</td>
<td>0,1,2,3,4,5i FDN</td>
<td>15</td>
<td>300</td>
<td>5,285</td>
<td>671,185</td>
<td>214,446</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lossBlk</td>
<td>r</td>
<td>1,2,3,4,5i FDN</td>
<td>14</td>
<td>280</td>
<td>4,920</td>
<td>573,148</td>
<td>186,554</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lossMeta</td>
<td>r</td>
<td>1,2,3,4,5i FDN</td>
<td>14</td>
<td>280</td>
<td>4,789</td>
<td>590,902</td>
<td>256,211</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cutBlk</td>
<td>r</td>
<td>1,2,3,4,5i FDN</td>
<td>14</td>
<td>280</td>
<td>4,982</td>
<td>619,235</td>
<td>194,904</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cutMeta</td>
<td>r</td>
<td>1,2,3,4,5i FDN</td>
<td>14</td>
<td>280</td>
<td>5,020</td>
<td>595,579</td>
<td>188,863</td>
<td></td>
</tr>
<tr>
<td>System (Sys)</td>
<td>panicDN</td>
<td>r/w</td>
<td>1,2,3,4,5i FDN</td>
<td>10</td>
<td>150</td>
<td>1,977</td>
<td>260,660</td>
<td>103,155</td>
<td></td>
</tr>
<tr>
<td></td>
<td>deadDN</td>
<td>r/w</td>
<td>1,2,3,4,5i FDN</td>
<td>10</td>
<td>150</td>
<td>1,751</td>
<td>228,550</td>
<td>90,693</td>
<td></td>
</tr>
<tr>
<td></td>
<td>readOnlyDN</td>
<td>w</td>
<td>1,2,3,4,5i FDN</td>
<td>5</td>
<td>75</td>
<td>368</td>
<td>61,944</td>
<td>27,820</td>
<td></td>
</tr>
<tr>
<td>Combination (COM)</td>
<td>Process (Proc)</td>
<td>rwrpc</td>
<td>1,2,3</td>
<td>3</td>
<td>210</td>
<td>7,037</td>
<td>237,318</td>
<td>111,247</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Network (Net)</td>
<td>rwrpc</td>
<td>1,2,3</td>
<td>3</td>
<td>210</td>
<td>6,539</td>
<td>219,602</td>
<td>103,192</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data</td>
<td>rwrpc</td>
<td>1,2,3</td>
<td>3</td>
<td>210</td>
<td>7,392</td>
<td>251,902</td>
<td>117,475</td>
<td></td>
</tr>
<tr>
<td></td>
<td>System (Sys)</td>
<td>rwrpc</td>
<td>1,2,3</td>
<td>3</td>
<td>210</td>
<td>7,056</td>
<td>235,379</td>
<td>110,519</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bug</td>
<td>rwrpc</td>
<td>1,2,3</td>
<td>3</td>
<td>105</td>
<td>3,847</td>
<td>53,011</td>
<td>29,656</td>
<td></td>
</tr>
<tr>
<td>Multiple (Mul)</td>
<td>AnarchyApe (AA)</td>
<td>rwrpc</td>
<td>1,2,3</td>
<td>3</td>
<td>480</td>
<td>17,244</td>
<td>602,512</td>
<td>280,556</td>
<td></td>
</tr>
</tbody>
</table>


\[a, i = 1, 2, \ldots, 10\]

by \(\text{tid}\). And \(T\) indicates the type of the relationship, such as a local function invocation or a remote procedure call (RPC). For example, Fig. 2a shows a trace tree of the process of handling a rename request in HDFS. The client first gets the information of the target file from the namenode, like whether the file exists, using the "RPC: getFileInfo", and then renames the file with the "RPC: rename". In this trace, the event set \(E = \{1, 2, 3, 4, 5\}\) and the relationship set \(R = \{a, b, c, d\}\), whose contents are shown in Fig. 2b.

Based on the events and the relationships, a trace can be reconstructed to a trace tree, where the nodes in the tree correspond to the events, and the edges correspond to the relationships. In a trace tree, a left node of brother nodes happens earlier than right nodes, e.g., event 2 happens earlier than event 3 in Fig. 2a. And we say a father node triggers a child node using a method of \(T\), e.g., edge \(a\) represents that the event 1 triggers the event 2 using the method of local function invocation. Naturally, we can convert a trace tree to a linear event sequence with the methods like DFS or using Call (C) and Return (R) to describe a node. For example, the trace tree of Fig. 2a can be expressed as "1, 2, 3, 4, 5" using DFS, or "C1, C2, C3, R1, R2, C4, C5, R3, R4, R5" using Call and Return method.

### 3.1.2 Trace Format in TraceBench

Traces collected by MTracer are organized in the form of the trace tree. The detailed information \(I\) of an event is defined as \(\{\text{name, st, et, ip, hname, a, d}\}\), representing the event name, start time stamp, end time stamp, host IP address, host name, agent, and description, respectively. The event name is the name of the event, e.g., function name. The start and end time stamps record when the event starts and finishes in nanoseconds, which can be used to calculate the execution time, or say latency, of the corresponding operation. The host IP address and host name denote where the event is from. The agent represents an inner part of the system, like a Java class. The description records the execution results, such as return values and exception information.

For efficiency and overhead consideration, we introduce \(\text{nid}\) rather than \(\text{eid}\) for an event [15], [19], [22]. Each time a thread communicates with another thread, a new \(\text{nid}\) is generated. After that, we can use the \(\text{nid}\) to distinguish each event and reduce the overhead.

Using events and edges, we can also reconstruct the trace tree. Fig. 3 illustrates the process of reconstructing the trace tree using the information of Fig. 2c, which contains four steps: (a) select all the events and edges with the same \(\text{tid}\) identifying the trace; (b) classify the selected events into classes according to their \(\text{nid}\), which means all the events in the same class are generated from the same thread; (c) calculate the relationships in each class using the start and end time stamps, e.g., because event 1 starts earlier and finishes later than event 2, and there is no other ancestor of event 2, event 1 is the father of event 2; and (d) construct the relationships between classes using edges, where the event identified by \(\{\text{fnid}, \text{fst}\}\) is the father of all the root nodes in the classes identified by \(\text{cid}\). Regardless of the database querying operation in step (a), the computational complexity of reconstructing a trace tree containing \(n\) events and \(m\) edges.
edges is $O(nm + n^2/m)$ on average, and $O(n^2)$ in the worst case. For reasonable traces, the computation is typically not expensive. For example, for a trace with 342 events and 151 edges in TraceBench, the reconstruction requires less than 50 ms on a 3.1 GHz Intel-based computer.

### 3.1.3 Some Examples

Traces collected by MTracer both record the normal behaviors of the monitored system and the abnormal behaviors when correctness problems or performance problems happen. Following, we give several examples selected from TraceBench.

Fig. 4a illustrates the normal process of reading a data block from HDFS, which is a part of handling a read request. The client first chooses the best datanode from the datanode list informed by the namenode, and then reads the block from the best datanode. However, when the best datanode encounters a killDN fault, which kills the HDFS processes on some datanodes, the operation of reading the data block will fail. Then, the client tries other datanodes in the datanode list. At last, the reading operation succeeds on a certain datanode or fails after trying all datanodes in the list. Fig. 4b clearly shows the process of finally succeeding after two failed tries. The content in the dashed box records the description fields of corresponding events, which depict the details of each try.

As another example, when encountering a fault of slowHDFS, which slows the whole network of the HDFS cluster, the latencies of some events in the trace tree would increase. For example, when the network slows down by 1 second, the latencies of event 2 and event 3 in Fig. 4a will increase by about 1 second, and thus event 1 by about 2 seconds. And the latencies of event 4 and 5 will not visibly change, since they happen inside a single host and have no relation with the network.

### 3.2 Structure

As shown in Table 1, TraceBench includes three classes: Normal, Abnormal, and Combination, respectively recording the information when the monitored system runs normally, meets a permanent fault, and encounters temporal faults. Each class consists of several types, which focus on different aspects of each class, e.g., the Datanode type mainly considers the aspect when the number of datanodes changes. In each type, we introduce different faults, workloads, and variables to simulate different scenarios. For example, in

![Fig. 4](image-url)
fault = panicDN
parameter = -R datanode024,datanode033,datanode029
inject time = 2013-11-13 10:11:55
recover time = 2013-11-13 10:17:47 //manual

Fig. 5. A fault record in a fault file, where the parameter defines the target datanodes, and the comment “manual” in the last line means the recovery is made by hand rather than automatically.

Datanode type, we introduce three workloads, and with each workload we vary the datanode number to be 1, 5, 10, . . . , 50, respectively, to simulate the scenarios with various workloads in different cluster scales. According to the fault, workload, and variable, a trace type contains many trace files, each of which corresponds to a certain scenario. In TraceBench, we name a set of traces according to the items of the first five columns in Table 1, i.e., “[Class][Type][Fault][Workload][Variable]?”. In the form of the regular expression, where the bold words means the sets of items appeared in corresponding columns of Table 1. In addition, abbreviations given in the brackets in Table 1 are used for compressing the names. As an example, the trace set named as Normal_Clientload_-_r, or NM_CL_r for short, contains the traces collected under the workload r in the Clientload type of Normal class. Following, we respectively introduce each trace class of TraceBench.

The traces in the Normal class record the behaviors when the HDFS system runs normally, without injecting any faults. The Normal class consists of Clientload type and Data-
node type. The Clientload type considers different speeds of workloads, and can be used to study the capacities of the HDFS system in dealing with different workloads in a certain scale (i.e., 50 datanodes). Since each client generates the workload in the same speed, we use the number of clients to represent the total workload speed of all valid clients in Table 1, and “1, 5, 10” means setting the number of clients to be 1, 5, 10, . . . , 50, respectively. In contrast, the Data-
node type fixes the speed of workload (i.e., 30 clients) and changes the number of datanodes, to study the behaviors with different system scales.

The traces in the Abnormal class are collected when a permanent fault is injected into the HDFS system with a fixed scale (i.e., 50 datanodes) and a fixed workload speed (i.e., 30 clients). The traces contain the information about the injected faults, and can be used to study the behaviors when the system runs abnormally. According to the types of faults, the Abnormal class can be divided into four types, i.e., Process, Network, Data, and System, which will be dis-
cussed later. Except the slowHDFS affects the whole network, all the faults in this class are injected into datanode(s), e.g., the lossBlk means delete all the data blocks in some datanodes. So, the FDN in Table 1 represents the datanodes with a fault injected.

When collecting the traces in Combination class, faults are randomly picked and injected into the HDFS system, and later recovered automatically or manually (See column Recovery in Table 3). Thus, the Combination class contains the traces collected both when the system runs normally and abnormally. In this class, together with a trace file, a fault file is also given, recording the information about the injected faults, including the fault name, injection time, recovery time, etc. Fig. 5 shows an example of an injected fault record. The Combination class consists of the Single type, in which faults are chosen in only one fault type, and the Multiple type, in which faults are picked from multiple fault types, e.g., selected from all the faults in AnarchyApe [27]. Note that, besides the faults in Abnormal class, we also employ several Bug faults in Single type, which are real-world bugs selected from Hadoop issues repository [28] and will be introduced later. In addition, in this class, we repeat the collection process of each scenario for three times, which results in three trace files, i.e., “1, 2, 3” in the column Variable. Since the faults are randomly picked and injected, the three trace files of the same scenario contain different faults and a same fault employs different parameters.

3.3 Collection Environment

TraceBench is collected in a real environment, which consists of more than 100 virtual machines (VMs) hosted on our IaaS platform, i.e., CloudStack [29]. Fig. 6 shows the envi-
nronment, which contains the following components:

- **HDFS**: providing a distributed storage service, containing 50 datanodes and one namenode.
- **Clients**: used to generate workloads to HDFS, to simulate the real usage of HDFS, containing 50 hosts.
- **MTracer server**: receiving, storing, and visualizing traces generated when HDFS processes the requests from clients.
- **Controller**: controlling the whole process of collection, and also being in charge of injecting faults.
- **Ganglia server** [4]: monitoring the whole environment, to help for solving the unexpected issues in collection, like VMs are shut down by accident.

The MTracer Server is deployed on a VM with 4 GB memory and 8 × 1 GHz CPU, while all the rest hosts are deployed on the VMs with 2 GB memory and 4 × 1 GHz CPU. The OS that all the VMs use is CentOS 6.3.

3.4 Process of Collection

We take the trace file as the unit of collection. Fig. 7 shows the processes of collecting trace files in each class. When collecting a trace file in the Normal class, we first start the MTracer server and the HDFS service with a certain number of datanodes, and then launch the workload on some clients concurrently. When finishing the collection, workload is stopped first. Then, the HDFS service and the MTracer server are shut down after waiting for a few minutes, to guarantee that all requests are finished smoothly and no
fragmentary trace is collected. Therefore, in Table 1, the items of the column Collection Time are composed of the running time and the waiting time of all contained trace files. When collecting trace files in the Abnormal class, faults are injected before starting the workload, and the recovery are done after handling all the requests, to ensure all the traces are collected when the system runs abnormally. In contrast, in the Combination class, a randomly chosen fault is injected and the system is later recovered after starting the workload, and the next fault acts in the same way after an interval, to simulate the occasionally occurring faults. In addition, except a few tasks, such as recovering the system from a System fault, most jobs of the trace collection are done automatically.

Some parameters used in the collection are described as follows. First, except the NM_DN set, the number of datanodes is set to 50 during collecting, corresponding to the maximal capability of the HDFS system in our environment. Except the NM_CL set, the number of clients is fixed to 30, producing an appropriate workload, which contains enough requests that can be processed by the HDFS service in time. Second, the collection time of a trace file in the Normal class is 60 minutes (50 minutes for running and 10 minutes for waiting), which is enough for collecting plenty of traces to reflect related statistical features. The collection time of an Abnormal file is set to 20 minutes (15 minutes for running and 5 minutes for waiting), since HDFS employs some mechanisms to detect and to avoid faults automatically, like the heart-beating protocol, and thus the number of the traces containing exception information may reduce significantly after 20 minutes. Finally, most step sizes of variables are 5, i.e., changing 10 percent each time comparing to 50, which balances between reflecting statistic features of HDFS and controlling the number of trace files, e.g., the numbers of datanodes are set to 5, 10, 15, . . . , 50 in NM_DN set.

3.5 Workloads

There are about 30 different HDFS requests [14], which can be mainly divided into three kinds: file read request, file write request, and metadata request. A read request downloads files from HDFS to local host, like copyToLocal, while a write request uploads files from local host to HDFS, like copyFromLocal. Both of them involve communicating with the namenode to get the information about data blocks, and

<table>
<thead>
<tr>
<th>Kind</th>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>copyToLocal</td>
</tr>
<tr>
<td>write</td>
<td>copyFromLocal</td>
</tr>
<tr>
<td>metadata</td>
<td>mkdir, touchz, mv, chmod, chown, ls, count, rmr</td>
</tr>
</tbody>
</table>

with some datanodes to download or upload data blocks. The metadata requests, such as rm and ls, operate metadata files, and only communicate with the namenode using RPCs without data block accessing. Table 2 lists the requests contained in TraceBench. The read and write request operates on the files that are automatically generated on HDFS and clients, respectively, with the number of contained data blocks spreading from 1 to 19. Eight most popular requests are considered in metadata kind, involving creating, modifying, querying, and removing of files and directories.

The workloads that contain only read, write, and metadata requests are denoted by r, w, and rpc, respectively. Besides, we also introduce two other workloads: rw, containing both read and write requests, and rwrpc, containing all the three. When collecting the traces in each trace type, we introduce different workloads as needed. For example, since the faults in the Data type do not influence the write requests, we do not introduce the w workload in the AN_Data set.

Bash scripts are employed on each client for generating workloads. Each script takes charge in one kind of workload, e.g., rpc.sh produces the rpc workload. Using a loop, requests with different parameters are sent to the HDFS service continuously by a client. All the requests from the clients form the global workload. Intervals are introduced between neighbouring requests in a script, i.e., after finishing a request, a script waits for a moment and then starts the next one, to control the speeds of requests.

3.6 Faults

As Table 3 shows, we introduce 17 faults of five types during trace collection. The faults are selected from AnarchyApe [27], a mature injection tool for Hadoop, and from Hadoop issues repository [28], which stores real-world bugs reported by users (Column Selected From). Some of these faults bring correctness exceptions when handling requests, e.g., killDN may cause a failure of reading a file, while others may introduce performance problems, e.g., HADOOP-6,502 results in a slow listing operation (Column Category).

The faults in AnarchyApe can be categorized into four types. The faults in the Process type affect the HDFS processes on HDFS nodes. For example, the fault killDN kills the HDFS processes on some datanodes, while suspendDN suspends some processes. Network faults bring anarchies to the network in the cluster, such as slowHDFS slows the whole network by milliseconds, and slowDN decreases the speeds of sending and receiving packets on some datanodes. The faults in Data introduce errors in the data blocks or the metadata files on some datanodes. For example, corruptMeta changes the values of some bits in the metadata files, and cutBlk removes parts of bits in the data blocks. The System faults introduce problems to the OSs of the HDFS nodes, such as making OSs to be panic, dead or read-only. Besides
Applications

To validate the usability and authenticity, we have employed TraceBench in several trace-oriented monitoring topics, include anomaly detection, performance problem diagnosis, and temporal invariant mining. We have also carried out an extensive data analysis on TraceBench, which validates the high quality of the data set.

4.1 Anomaly Detection

The same kind of user requests usually result in the traces with similar topologies, which can be extracted as the properties that are useful in anomaly detection, problem diagnosis, and other topics.

For example, when reading a file, the client downloads the data blocks of the target file from HDFS one by one, where a data block downloading starts with invoking the function “blockSeekTo” (short for B), and ends with calling “checksumOk” (K) if success. If any data block fails in being downloaded after some retries, the whole request aborts and fails. In other words, in a successfully handled read request, the last data block should be downloaded successfully, which can be expressed by the Linear Temporal Logic (LTL) [30] as the following property:

\[ \circ B \land (\Box (B \rightarrow \Box (\Box \neg B)) \rightarrow \Box (\circ K)) \]

where the operators \( \Box, \circ, \land, \rightarrow, \text{and} \neg \) represent for All, Exist, Next, And, Imply, and Not, respectively. Therefore, \( \circ B \) expresses that at least one reading operation exists, \( B \rightarrow \Box (\Box \neg B) \) means no more data blocks are read after current block, i.e., the last reading operation, and \( \circ K \) means there exists a successful reading operation. Thus, the whole property means the request reads at least one data block and succeeds in the last reading, which defines a successful read request. If a trace violates this property, we say a failure happens.

Similarly, we also extracted the properties for write and metadata requests. To validate these properties, we checked the traces in the \( \text{AN} \) set in the form of SQL queries to detect failures. All of the failed requests are picked out correctly. Besides the properties for detecting failures, we have also extracted tens of properties for detecting various problems, such as datanode invalid, data block missing, and operation latency anomaly. All of these properties can be in turn used to monitoring an HDFS system with different methods, e.g., the Monitoring Oriented Programming (MOP) [31] framework in runtime verification (RV) [32], which is part of our future work [33].

4.2 Performance Problem Diagnosis

Principal component analysis (PCA) is widely used in analyzing traces for diagnosing performance problems. We implemented a PCA-based diagnosis method [34], [35], which finds the traces with abnormal latencies, and further locates the root causes of the problems. This method performs on trace clusters, where the traces in a cluster contain a
same call sequence, e.g., the call sequence of Fig. 2a is "fs
-mv"→"RPC: getFileInfo"→"getFileInfo"→"RPC: rename"→
toke Call sequence as the cluster length. We call the number of traces contained in a cluster as the cluster capacity, and call the number of events in the corresponding call sequence as the cluster length.

Using the format of linear event sequence, we partitioned traces in COM_Mul_All_rwrpc set into clusters according to their call sequences, and then evaluated the method in the aspect of finding abnormal traces with these trace clusters. Table 4 shows the experiment results, which contains two parts: 1) the results of the trace clusters with a same length (i.e., 7) and with different capacities, and 2) the results of the clusters with a same capacity (i.e., 7) and with various lengths. For each cluster, we ran the method for five times and report the average time as the analysis time in row Time. In addition, the experiments are carried out on a VM with 8 × 1 GHz CPU and 4 GB memory.

As Table 4 shows, this diagnosis method finds all abnormal traces in some cases, and however sometimes only a small part. Actually, the results depend on the features of data, since the thresholds used for judging the abnormal traces are calculated from the traces. So, when some very abnormal traces exist, some other less abnormal ones would be ignored due to the affected thresholds. For example, only 1 of 30 abnormal traces was found in the trace cluster with the capacity of 400 in Table 4, and after removing this one, another 27 abnormal traces were correctly picked out. In addition, according to the results, this diagnosis method is pretty accurate, with no false alarm in our experiments.

On the other hand, the analysis time increases very fast when the cluster length increases, but pretty slow when the cluster capacity grows, which indicates that this method is more sensitive to the length rather than to the capacity. The primary cause is the process of calculating eigenvalues and eigenvectors of a square matrix for getting principal components, where the calculating time is mainly related to the matrix size that exactly equals to the cluster length and has no relation with the capacity. Therefore, this method seems to be more applicable for short traces. Parallel approaches [36] or a segmentation based method [37] may be helpful for improving the efficiency.

### 4.3 Temporal Invariant Mining

As an important aspect of system features, temporal invariants record the rules of the orders obeyed by two system operations, which can be obtained by mining system logs and can be used to understand systems, detect abnormal behaviors, diagnose deadlocks, infer higher-level properties, etc. Based on TraceBench, we evaluated a mining algorithm in Ref. [38] with the corresponding tool Synoptic [39] (revision id: cce864f389c7). After formatting the traces both into the totally ordered (TO) format and the partially ordered (PO) format (refer to [38]), we have mined plenty of invariants in different trace sets with various request kinds in the NM_CL set. Table 5 shows the numbers of mined invariants, where →, ←, and ⊥ respectively means the invariant type of Always followed by, Always precedes of, and Never followed by (refer to [38]). Both TO logs and PO logs imply useful invariants. For example, INITIAL → blockSeekTo and blockSeekTo → checksumOk mined from the NM_CL_r_1C set, respectively mean each read request contains at least one data block reading operation and each successfully handled reading operation invokes the function “checksumOk”, which coincide to the property introduced in Section 4.1. Actually, many properties can be inferred based on these invariants.

When dealing with PO logs, Synoptic treats the same kind of events generated from different hosts as different events, which makes it possible to mine more useful invariants for concurrent systems. However, this also has several limitations. First, too many invariants are generated due to the massive event types, e.g., more than 10,000 in the NM_CL_r_1C set, which makes it difficult to search desirable features. Second, since the occasionally happened or occasionally not happened event combinations between hosts are treated as inherent system features, some false invariants arise. For example, the invariant receive Block_datanode001 ⊥ receive Block_datanode006 mined from the NM_CL_w_1C set, (where the function “receiveBlock” running on datanodes receives a copy of data block from the client) represents that there does not exist the situation in the NM_CL_w_1C set that HDFS stores a data block on datanode006 after storing on datanode001, which is still possible in other executions. Third, many invariants contain the same information, and we call such invariants as identical invariants. For example, the invariants Conn_datanode001 → receive Block_datanode001, ..., Conn_datanode007 → receive Block_datanode006 mined from the NM_CL_w_1C set (where “Conn” is short for “OP: connect next Datanode”) all mean that when receiving a copy of data block, the related datanode first builds connection and then receives the copy. To summarize, when mining temporal invariants in PO logs, Synoptic seems to be more suitable for the systems with few hosts.

We consider a postprocess on the generated invariants is helpful for improving Synoptic. For example, using techniques in machine learning and data mining, the identical invariants can be effectively eliminated, like summarizing the 50 identical invariants in above example into one invariant, i.e., Conn_host → receive Block_host.
4.4 Data Analysis

We have also carried out an extensive data analysis based on TraceBench and get some characteristics of HDFS in statistics, which can also validate the data set. In this section, we give parts of our analysis results.

4.4.1 Requests Handling

With the traces in the Normal class, we have analyzed the relationships between the request handling speed and the request receiving speed of HDFS. Since all the clients take the same speed of sending requests, we use the number of clients to represent the request receiving speed. Because the time of handling a read or write request is related to the size of the file, we use the speed of downloading or uploading instead of the request handling speed. Besides, we use the number of datanodes to represent the scale of the HDFS system.

Theoretically, an HDFS service in a specific environment has a certain upper limit in handling, i.e., the maximal number of the requests that can be processed in a period, which reflects the request handling ability of HDFS. If the request receiving speed is smaller than the handling limit, all the requests can be handled in time. Otherwise, the HDFS service would be overloaded and some requests may be delayed or ignored.

Fig. 8 displays some analysis results. The curve with crosses in Fig. 8a represents the observed reading speeds under different request receiving speeds, where each point corresponds to a trace file in the NM_CL_r set. The solid curve is the perfect reading speed without considering the handling limit of HDFS. As the client number increases, the speed of reading increases almost linearly at first, which is close to the expected speed, and then increases slowly when there are more than 30 clients, due to the handling limit.

The reading speed in Fig. 8a is calculated with respect to the number of the events with a name of “OP: receive block” in a certain period, since we know the size of each block is 64 MB. The curve with squares in Fig. 8a is the network consumption monitored by Ganglia, which is resource-oriented. From Fig. 8a, we can observe that the reading speed is a little smaller than the network consumption. The reason is that other network packets are also counted, such as the events of MTracer and Ganglia. In addition, the match of these two curves implies the viewpoint of that traces also contain some information about resource usages.

The handling limit of HDFS is related to the cluster scale. Fig. 8b is extracted from the NM_DN_r set with a fixed reading speed, i.e., 30 clients. The solid line represents the perfect speed of reading with all read requests being processed in time. The curve with crosses displays the actual reading speeds. The reading speed increases first with the increase of datanode number, because the ability of the HDFS service is improved with a larger scale, and then holds steady around the theoretical speed when reaching 30 datanodes, which indicates that the handling limit of the service exceeds the requests receiving speed.

4.4.2 Workload Balancing

Based on the NM_CL_r set, Fig. 9 shows the results of analysis of workload balancing in handling read requests, where each point represents the percentage of the data block reading operations happened on a certain datanode in a trace file. For example, the crosses surrounded by the dashed rectangle represent the percentages of reading operations accomplished by datanode021 in each trace file. All points are distributed around the solid line, i.e., 2 percent, which means one of 50 datanodes accomplishes about 1/50 of the total tasks. This phenomenon also exists in handling write requests, which becomes even more obvious since the number of writing operations in NM_CL_w is more than the number of reading operations in NM_CL_r. Therefore, HDFS well balances the requests to all datanodes, which improves the resource utilization and avoids overloading.

Fig. 8. Analysis of requests handling speeds (a) under different request speeds and (b) with different cluster sizes.

Fig. 9. Analysis of workload balancing in handling read requests.
4.4.3 Correctness Problems

Faults may result in correctness errors and even failures in handling user requests. For example, fault killDN would cause a failure in uploading a data block, i.e., an error in a write request, and several such errors on the same data block will result a failure of the whole request. We can always judge if a trace in TraceBench contains any correctness errors or failures, by exploring the topology of trace tree and the description fields of contained events, like Fig. 4 shows.

Based on the AN_Proc_killDN_w set, Fig. 10 shows the relationships between the number of FDNs and the percentage of errors and failures. The curve with crosses is the percentage of failed block writing operations with different number of FDNs, and the solid curve displays the theoretical percentage values. These two curves are coincident, which validates the authenticity of our data. The curve with squares is the percentages of the failed write requests, from which we can conclude some statistical features, e.g., 80 percent of the datanodes should be valid if we do not want the percentage of the failed write requests to exceed 20 percent.

4.4.4 Performance Problems

Besides correctness problems, faults may also lead to performance problems, i.e., increase of the latencies rather than the topological changes of trace trees. Fig. 11 is plotted based on the AN_Net_slowHDFS set, and shows the average time of reading a block, writing a block, and RPC invocations, when some slowdowns are introduced into the network of the HDFS cluster. The processing time of each operation increases as the network becomes slower. The slope of each trend line reflects the sensitivity of each operation with this fault. Because requiring plenty of network communications, writing a block is the most sensitive one. According to the trace data, the slowdown of writing operation is about 1,000 times larger than the network slowdown, i.e., 1 millisecond slowdown in network results in about 1 second increase when writing a block. RPC is the least sensitive operation, where 1 millisecond slowdown results in about 1 millisecond increase, since only a few network interactions are required in an RPC invocation.

5 Threats to Validity

Following aspects may threaten the validity of TraceBench.

5.1 Collection System

We collected TraceBench with MTracer for the reasons mentioned in Section 2. MTracer stores the trace in the format of the trace tree, which can express sequential, concurrent, and recursive call-reply patterns [15]. However, this format is insufficient for expressing some other behaviors, like a single event causally depends on multiple father events [15], which may make TraceBench invalid in certain applications, e.g., the analysis methods in Pip [7] and Spectroscope [10]. A solution of this limitation is to adopt the format of DAG [15].

For efficiency consideration, MTracer can not perceive the errors and failures in traces. Therefore, there are no explicit tags existing in traces for indicating whether the trace is normal or not. However, like Fig. 4 and Section 4.4 shows, we can always exactly estimate a trace from the recorded information, such as the trace topology, the corresponding fault file which describes the injected faults, and the logged information in events, like the description field and latency field.

5.2 Monitored System

We collected our traces only on HDFS, which is a widely used cloud storage system in academia and industry. Many mechanisms in HDFS, like RPC, are commonly used in other systems. And we consider various scenarios of HDFS during collection. Therefore, TraceBench is representative and we believe TraceBench is helpful for many trace-oriented monitoring topics and other fields.

Our HDFS system is instrumented manually, which leads to record accurate execution paths of user requests. To reduce the monitoring overhead, we only instrument the key steps of requests, which record the main processes of handling HDFS requests. Specifically, we are more interested in the communications between hosts than the actions inside a single host. However, this may make TraceBench unsuitable for certain scenarios, like concerning about the details of operations inside a host.

5.3 Collection Environment

During trace collection, the HDFS cluster contains 50 datanodes, which is smaller than the production systems [40]. However, the cluster size of our HDFS is large enough to exhibit various features of HDFS for research purpose, since a user request in HDFS always involves limited datanodes. Moreover, it is really difficult for academic to deploy a very
large-scale cluster in practice. On the other hand, the number of the clients is also set to 50, equalling to the number of datanodes, which should be more in real world. However, the workloads generated from these clients are pretty close to the reality and can make the HDFS system in different states, like reaching the upper limit.

Besides the faults we inject, many others exist in real systems, like dropping packets in network. Nevertheless, the faults we choose are mainly from a mature injection tool created by Yahoo! for Hadoop clusters, and we employ all the fault types in this tool. Hence, we believe that the faults we inject cover the most frequent and representative faults in practice.

5.4 Supported Applications

Our data set can be used in many applications as Section 4 shows, however, also may fail in supporting some others. To help for determining if TraceBench is suitable for a certain application, we give some features of TraceBench as follows:

- Events are totally ordered in each trace;
- Does not assume global clocks;
- Concurrency exists between traces, but no concurrency exists in a single trace;
- No much attentions are paid to the actions inside a single host;
- Traces do not contain synchronization points [15], i.e., each event at most has one father event.

In addition, on the homepage of TraceBench, we list many applications that can or cannot be supported by our data set.

6 RELATED WORK

We have described the monitoring systems in Section 2.1, and in this section, we mainly introduce the related data sets. Currently, there exist some public data sets collected from multiple hosts, just like TraceBench. However, almost all of these data sets are resource-centric, job- or service-centric, which seldom involve in the details of system running.

Resource-centric data sets record the availability of hosts and components or the usages of resources, focusing on external, rather than internal, information of system programs. For example, the Failure Trace Archive (FTA) [41], [42], collected from parallel and distributed systems, records the information about resource failures; the Computer Failure Data Repository (CFDR) [43] provides the failure data in supercomputers and clusters, e.g., the storage failure data; and the Repository of Availability Traces [44] contains the events that indicate whether and when a host is available.

Job-centric data sets give a coarse-grained view of system running, such as when a job starts and finishes, which hosts are involved, without the details of how a job works. For example, the Grid Observatory (GO) [45], collected on a grid infrastructure, includes many data sets that record the information around jobs, e.g., jobs lifecycle; the Parallel Workloads Archive (PWA) [46] provides job-level usage data and is widely used in the research of job scheduling strategies for parallel systems; the Google Cluster Data [47] describes hundreds of thousands of jobs, composed by lots of tasks, together with the task resource usages.

Service-centric data sets focus on the aspects of services provided by systems for supporting service computing [48] topics. For example, the PW dataset [49], crawled from a service registration center, records implicit feedback of real-world web services; the Web Service QoS Dataset [50], [51] contains plenty of real-world web service Quality of Service (QoS) records by monitoring abundant web services, for validating QoS-based approaches, like Ref. [52]; the CLUS evaluation dataset [53] includes reliability data collected from multiple web services deployed on different geographical locations, which can be used in predicting the reliability of services.

In contrast, TraceBench records the fine-grained information about handling user requests, i.e., the details and casual relations in user requests involving multiple hosts, and hence is user request-centric. To the best of our knowledge, TraceBench is the first fine-grained user request-centric open trace data set. Besides, TraceBench also exceeds some data sets in certain aspects. For example, the CFDR lacks normal data, and the Repository of Availability Traces has limited information besides availability.

7 CONCLUSION AND FUTURE WORK

In this paper, we provide a fine-grained user request-centric open trace data set, called TraceBench, collected with a trace-oriented monitoring system we have developed. TraceBench is collected in a real environment considering different scenarios, involving the cluster scale, request type, workload speed, injected fault, etc. We employ TraceBench in several applications to validate the usability and authenticity, and the results show that TraceBench is helpful for the trace-oriented monitoring research topics, such as anomaly detection, performance problem diagnosis, and temporal invariant mining. Moreover, due to the high quality of TraceBench, other related research, like system understanding, can also employ our data set.

In the future, we plan to: 1) accomplish the aforementioned two trace-oriented monitoring techniques, i.e., the runtime verification based anomaly detection method and the segmentation based performance problem diagnosis method; 2) explore more applications on TraceBench to validate our data set and to develop new methods; 3) if needed, collect some more trace sets with a more general trace format, like DAG, and on some other systems, e.g., MapReduce.

ACKNOWLEDGMENTS

The authors are grateful to the anonymous reviewers and the editors for their careful review of the manuscript and for their valuable comments that improved this work. The authors also would like to thank Halbo Mi, Yongquan Fu, Ivan Beschastnikh, Siqi Shen, and Hengbiao Yu for the advises of our work. This work is supported by the National 973 Program of China under the Grant No. 2014CB340703, the National Natural Science Foundation of China under the Grant No. 61472440, No. 61272140, and No. 61303064, and the Guangdong Natural Science Foundation with the Project No. 2014A030313151.
Zibin Zheng received the PhD degree from the Chinese University of Hong Kong, Hong Kong, China, in 2011. He is an associate research fellow in the Shenzhen Research Institute, the Chinese University of Hong Kong, Shenzhen, China. He received the ACM SIGSOFT Distinguished Paper Award at ICSE 2010, the Best Student Paper Award at ICWS 2010, First Runner-up Award at IEEE Hong Kong Postgraduate Research Paper Competition, and the IBM PhD Fellowship Award 2010-2011, etc. He served as a PC member of conferences such as CLOUD 2009, CLOUDCOMPUTING 2011, SCC 2012, ICSC 2012, and served as a reviewer for international journals such as SCP and JSS. His current research interests include service computing, cloud computing, and software reliability engineering. He is a member of the IEEE.

Michael R. Lyu received the PhD degree from the University of California, Los Angeles, in 1988. He is a professor in the Department of Computer Science and Engineering, the Chinese University of Hong Kong, Hong Kong, China. He initiated the First International Symposium on Software Reliability Engineering (ISSRE) in 1990. He served as a chair or co-chair for many conferences, such as ISSRE 2001, WWW 2010, SCC He 10, DSN 2011. He has been frequently invited as a keynote or tutorial speaker to conferences and workshops in the U.S., Europe, and Asia. His current research interests include software reliability engineering, distributed systems, service computing, information retrieval, social networks, and machine learning, and he has published more than 400 refereed journal articles and conference papers in these areas. He is a fellow of the IEEE and the AAAS for his contributions to software reliability engineering and software fault tolerance.