Information and Software Technology 119 (2020) 106234

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

How are distributed bugs diagnosed and fixed through system logs?[☆]

Wei Yuan[a,b,c], Shan Lu[b], Hailong Sun[a,c,*], Xudong Liu[a,c]

[a] SKLSDE Lab, School of Computer Science and Engineering, Beihang University, Beijing, China 100191
[b] University of Chicago, Chicago, USA
[c] Beijing Advanced Innovation Center for Big Data and Brain Computing, Beijing, China 100191

A R T I C L E I N F O

Keywords:
Software bug analysis
Distributed systems
Issue reports
System logs

A B S T R A C T

Context: Distributed systems are the backbone of today's computing ecosystems. Debugging distributed bugs is crucial and challenging. There are still many unknowns about debugging real-world distributed bugs, especially through system logs.

Objective: This paper aims to provide a comprehensive study of how system logs can help diagnose and fix distributed bugs in practice.

Method: The study was carried out with three core research questions (RQs): How to identify failures in distributed bugs through logs? How to find and utilize bug-related log entries to figure out the root causes? How are distributed bugs fixed and how are logs and patches related? To answer these questions, we studied 106 real-world distributed bugs randomly sampled from five widely used distributed systems, and manually checked the bug report, the log, the patch, the source code and other related information for each of these bugs.

Results: Seven findings are observed and the main findings include: (1) For only about half of the distributed bugs, the failures are indicated by FATAL or ERROR log entries. FATAL are not always fatal, and INFO could be fatal. (2) For more than half of the studied bugs, root-cause diagnosis relies on log entries that are not part of the failure symptoms. (3) One third of the studied bugs are fixed by eliminating end symptoms instead of root causes. Finally, a distributed bug dataset with the in-depth analysis has been released to the research community.

Conclusion: The findings in our study reveal the characteristics of distributed bugs, the differences from debugging single-machine system bugs, and the usages and limitations of existing logs. Our study also provides guidance and opportunities for future research on distributed bug diagnosis, fixing, and log analysis and enhancement.

1. Introduction

1.1. Motivation

Debugging is one of the most time consuming and expensive activities in software development [1]. Particularly, debugging distributed systems has become increasingly critical and challenging. On the one hand, distributed systems, including distributed computing infrastructures, distributed file systems, and others, have become the backbone of our computing community. People expect high availability of distributed systems, given the geo-diversity and inherent redundancy of distributed systems. A recent report shows that the outage of a top cloud provider in the US for a couple of days would lead to more than 10 billion dollars' economic losses [2]. On the other hand, different from single-machine systems, bugs in large-scale distributed systems could have root causes, error propagations, and/or failure symptoms going beyond one node, which we refer to as *distributed bugs*. Debugging these distributed bugs poses great challenges to distributed system users and developers [3–7], thus calls for new efforts to help tackle the problems.

Fig. 1 illustrates a real-word distributed bug (mr5488) in Hadoop MapReduce. In MapReduce, after an application is (1) submitted to ResourceManager (RM) and (2) gets executed by an ApplicationMaster (AM), its status on AM will be (3) periodically polled by its owner client. At some point when the AM crashes, both the RM and the client will learn that. Then (4) the RM will launch a new AM on another node to recover and continue the application, and (5) the client will obtain the address of the new AM from the RM. Unfortunately, sometimes the client may get the old AM's address, if the new AM's launching is unexpectedly slow. As a result, the client will fail to connect to the AM, and report that the application has failed in the log, even though the application is successfully completed by the new AM.

☆ This work was mostly done when the first author visited University of Chicago.
* Corresponding author at: SKLSDE Lab, School of Computer Science and Engineering, Beihang University, Beijing, China 100191.
E-mail addresses: cindyyuanwei@buaa.edu.cn (W. Yuan), shanlu@uchicago.edu (S. Lu), sunhl@buaa.edu.cn (H. Sun), liuxd@act.buaa.edu.cn (X. Liu).

https://doi.org/10.1016/j.infsof.2019.106234
Received 13 June 2019; Received in revised form 29 September 2019; Accepted 16 November 2019
Available online 20 November 2019
0950-5849/© 2019 Elsevier B.V. All rights reserved.

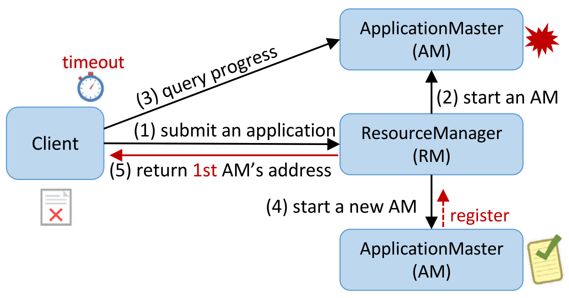


Fig. 1. A motivation example (mr5488).

Debugging this bug is challenging because its triggering and root cause involve multiple nodes, including the client, the RM, the old and the new AMs. The failure symptom of this bug is also non-trivial: the crash of the old AM is not a failure; the application failure observed by the client and reflected in the log also does not show the full picture of the bug symptom. As a matter of fact, even though the bug reporter provided detailed production-run failure logs from the very beginning, it still took developers three weeks to figure out the root cause.

In practice, debugging distributed bugs heavily relies on manual efforts, particularly manual checking of software logs. Unfortunately, debugging distributed systems through logs often faces unique challenges. For instance, a log entry that indicates fatal node failures or errors may not correspond to any system misbehavior due to the redundancy nature of distributed systems; the node where the system misbehavior is logged may not be the node where the root cause occurs. As we discussed in mr5488, the error message about the first AM's crash in the log actually is not a system misbehavior and will/should not be perceived by end users. At the same time, no log entries provide information about the incorrect AM address received from the RM. In addition, logs interleaved across different nodes due to the concurrent requests/threads will increase the difficulty of log analysis [8,9].

Although recent work has looked at how to automate the debugging process of some distributed system failures through automated log enhancement and log analysis [4,8,10–12], there are still many unknowns about debugging a wide variety of real-world distributed bugs, including locating failure occurrences, diagnosing root causes, and generating patches, as well as the challenges and opportunities presented by existing real-world software logs in conducting the above steps. Therefore, a comprehensive empirical study is highly desirable. Specifically, three core research questions and 12 sub-research questions are explored:

- RQ1: How to identify failures in distributed bugs through logs?
  - RQ1.1: Is the failure occurrence of a distributed bug reflected by logs?
  - RQ1.2: Can log analysis automatically tell the failure occurrence of a distributed bug?
  - RQ1.3: Is the failure type of a distributed bug reflected by logs?
  - RQ1.4: What log levels tend to reflect the failures of distributed bugs?
  - RQ1.5: Are distributed bugs' symptoms distributed?
- RQ2: How to find and utilize bug-related log entries to figure out the root causes?
  - RQ2.1: Are log entries sufficient for root-cause diagnosis?
  - RQ2.2: How to find bug-related log entries for debugging?
  - RQ2.3: How to figure out root causes based on bug-related log entries?
- RQ3: How are distributed bugs fixed and how are logs and patches related?
  - RQ3.1: How are distributed bugs fixed?
  - RQ3.2: Are logs changed in bug patches?
  - RQ3.3: What is the relationship between logs and patches?
  - RQ3.4: How quickly are distributed bugs fixed?

Note that understanding how distributed bugs are diagnosed in the fields is very challenging. Although previous work [5,13,14] has studied the symptoms and causes of many bugs in open-source distributed systems based on bug reports in JIRA bug databases, previous methodology does **not** apply here. First, many bugs in distributed systems actually have their root causes and error propagations completely limited in one node, and hence are *not* the target of this study. Second, most bug reports in JIRA bug databases do *not* contain run-time log information. It is hence impossible to study how logs are useful to diagnosing these bugs.[1] Third, although failure symptoms and bug root causes are often explicitly mentioned in bug reports, how exactly the diagnosis was conducted is often *not* explained in details and hence requires much manual effort in studying bug reports, system source code, patches, and others.

1.2. Contributions

In this work, we carefully studied 106 real-world distributed bugs and the run-time logs associated with their bug reports, randomly sampled from five widely used distributed systems including Cassandra, HBase, HDFS, MapReduce, and ZooKeeper. Our study provides seven interesting findings and the corresponding implications about how developers identify failure occurrences, diagnose root causes, and produce patches for distributed bugs, as well as the role of software logs during this process. The observations in our study identify unique challenges in debugging distributed bugs comparing with traditional single-machine software bugs. Meanwhile, our study points out the opportunities and provides guidance to future research on diagnosing and fixing distributed bugs, as well as enhancing and analyzing distributed system logs. Here, we highlight our major findings and their implications for future research below.

For only about half of the distributed bugs, the failure occurrences are indicated by FATAL or ERROR log entries. Meanwhile, FATAL log entries are not always fatal, and INFO log entries could be fatal, and this mismatch is closely related to the unique failure symptoms (e.g., inconsistency and global symptom) of distributed bugs. Our study finds that most (93%) of bugs' failure occurrences are reflected by existing logs, in the format of exceptions, abnormal log repetition, inconsistent log entries, etc. However, existing software logs are far from perfect: some bugs' failure occurrences cannot be reflected, or are even misleadingly reflected by logs; failure-indicating log entries are **not** assigned with high-importance tags — their log-level tags are almost equally distributed among FATAL, ERROR, WARN, and INFO; the exact failure types cannot easily be determined from logs, particularly for resource leak and data loss failures. Our study also sheds light on the opportunities and challenges in automatically analyzing logs to identify failure occurrences and failure types. The details are presented in Section 3.

For more than half of the bugs, root-cause diagnosis relies on log entries that are not part of the failure symptoms. Our study finds that although existing logs provide sufficient information for developers to figure out the root causes of most bugs (89%), it is **non-trivial** to locate all the useful log entries. In fact, for more than one third of the studied bugs, one has to go beyond the node(s) where the failure symptom is observed to find log entries that help point out the failure root causes. The details are presented in Section 4.

The root causes of a significant portion of our studied distributed bugs are not fixed by the patches — the distributed nature of these bugs often makes fixing root causes impractical. We thoroughly studied the patches for all the bugs, and found that *one third* of bugs are fixed by preventing specific failure symptoms **without** eliminating bug

[1] Repeating a real-world bug in distributed systems is extremely time consuming, as most JIRA bug reports do not contain detailed bug-triggering inputs and environment information. For this reason, almost all previous bug studies do not repeat bugs [5,13,15–17].

Table 1
Bug Sampling.

App.	# CBS Bugs	Contain Log	Contain Patch	Distributed Bugs	Sampled
Map**R**educe	517	138	136	84	26
HDFS	559	94	93	31	15
HBase	958	239	239	72	24
Cassandra	864	234	230	111	33
Zoo**K**eeper	134	42	42	26	8
Total	3032	747	740	324	106

* The bold texts are used to denote each system in bug IDs.

root causes or intermediate errors. These patches typically rely on three strategies including operation retries, exception ignoring, and early error/failure exposing, and are promising to be automated in the future. Furthermore, the patches go beyond the symptom node(s) for *one third* of the bugs, and go beyond the exception call stacks for almost *half* of the times. These all pose challenges to fixing distributed bugs. The details are presented in Section 5.

We also released our in-depth analysis of each bug with the paper[2] for the community to use.

The rest of this paper is organized as follows. Section 2 describes the methodology adopted in our study. Section 3 shows how to identify the failures through logs for the studied distributed bugs. Section 4 presents how to figure out root causes with the help of logs. In Section 5, we discuss different fixing strategies, and the relationship between logs and patches. Section 6 highlights the main findings and summarizes the actionable results and future research opportunities. Section 7 discusses related work, followed by Section 8 concluding this paper.

2. Methodology

2.1. Basic definitions

Failure, Error, and Fault These three terms are repeatedly referred to in this paper. We use their standard definitions in system dependability field [18]: "A system *failure* occurs when the delivered service deviates from fulfilling the system function. An *error* is that part of the system state which is liable to lead to subsequent failure. The adjudged or hypothesized cause of an error is a *fault*." The fault could be a hardware problem (e.g., network partitions or node power outage) or software code design/implementation defects.

Distributed bugs We consider a bug to be *distributed*, if its error propagation, which starts from the initial fault and ends at a system failure, involves more than one node.

Log symptoms Log entries that reflect system failures are referred to as failures' log symptoms.

2.2. Distributed systems under study

Our goal is to understand how distributed bugs are diagnosed and fixed in large-scale distributed systems. To achieve this goal, we select five widely used open-source distributed systems that cover different functionalities: Hadoop[3] MapReduce (distributed computing framework), HDFS (distributed file system), HBase[4] (distributed key-value store), Cassandra[5] (distributed key-value store), and ZooKeeper[6] (distributed configuration and synchronization services). These are all big Java applications (144K–3M lines of code), with long development history (11–15 years), and mature bug datasets, which we describe below.

[2] https://cindyyw.github.io/DBugSet/index.html.
[3] http://hadoop.apache.org.
[4] http://hbase.apache.org.
[5] http://cassandra.apache.org.
[6] http://zookeeper.apache.org.

2.3. Bugs under study

To identify real-world distributed bugs to study, we started with all the *fixed* bugs contained in the open source cloud bug suite (CBS) [13] (as shown in Table 1). This dataset contains over 3000 "major" issues in the corresponding JIRA bug databases of the above five cloud systems over a period of three years (1/1/2011-1/1/2014), and it has also been widely used by existing works, such as [5,19] and [20]. We filtered out bugs that do not contain run-time logs, as it would be difficult to figure out how exactly a bug was diagnosed by developers otherwise, or patches in their bug reports, as it would be difficult to figure out the exact fixing strategy otherwise. Next, we carefully examined *each* of the bug reports that contain logs and patches, and identified 324 of them as *distributed* bugs (based on the definition of distributed bugs in Section 2.1). Finally, we randomly sampled 106 of them (about 30% of the identified distributed bugs (the fifth column in Table 1) for each project) for in-depth bug diagnosis and fixing study. The table also lists the number of bugs that remain after each step of filtering. The size of our studied dataset is comparable with existing efforts on bug study of distributed systems [5,21,22], but all the bugs we studied are distributed bugs (i.e. the corresponding error propagation of a bug involves multiple nodes) containing log information, not general bugs of distributed systems. More recent distributed bugs from today's issue tracking systems can be considered in future study, to further confirm that the time period does not have any effect on the results in our study. As the distributed nature does not change in the target systems, thus the characteristics discovered and discussed in our study still apply.

2.4. Content inside logs

A big part of our study is to analyze the logs in the bug reports and see how developers (or anyone can) used it for failure diagnosis and bug fixing. All the five target systems use Apache Log4j[7] to record their logs. We provide some background about these logs below, and we will refer to each log record as a *log entry* in this paper.

Fig. 2 illustrates different elements of a log entry.

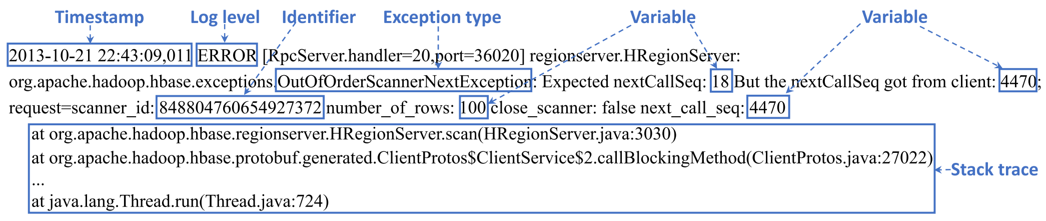
Time Every log entry includes an explicit timestamp that indicates when this entry is recorded. Furthermore, timing/ordering information is also embedded in other format, such as the server startcode (timestamp of server startup) in HBase, the transaction id (zxid) in ZooKeeper, etc.

Log Level Developers assign a log level to almost every log record, indicating how important the log entry is. Log levels in our studied systems include, ranking from high (more important) to low (less important), FATAL, ERROR, WARN, INFO and DEBUG (DEBUG log entries are only enabled during debugging).

Identifier (ID) The studied systems often give unique IDs for nodes, jobs, intra-job tasks, system components, etc.

Variable This refers to the value of the variable which is defined in the program and output in the log entry, including object size, object state, sequence number, etc.

[7] https://logging.apache.org/log4j.



```
2013-10-21 22:43:09,011 ERROR [RpcServer.handler=20,port=36020] regionserver.HRegionServer:
org.apache.hadoop.hbase.exceptions.OutOfOrderScannerNextException: Expected nextCallSeq: 18 But the nextCallSeq got from client: 4470;
request=scanner_id: 8488047606549273372 number_of_rows: 100 close_scanner: false next_call_seq: 4470
    at org.apache.hadoop.hbase.regionserver.HRegionServer.scan(HRegionServer.java:3030)
    at org.apache.hadoop.hbase.protobuf.generated.ClientProtos$ClientService$2.callBlockingMethod(ClientProtos.java:27022)
    ...
    at java.lang.Thread.run(Thread.java:724)
```

Fig. 2. An example log entry and its elements.

Table 2
An overview of provided log entries (short as *entries*).

Log Level	# Log entries	# Bugs containing these entries	# Entries with exceptions	# Exceptions in log symptom	# Irrelevant exceptions[8]
FATAL	17	14	13	11	2
ERROR	78	53	66	53	3
WARN	78	37	30	16	4
INFO	501	75	24	12	3
DEBUG	189	27	5	4	1

Exceptions that are not related to any bugs.

Exception In most cases, an exception is handled by a pre-defined catch block (i.e., *exception handler*), and printed, including exception-type and stack-trace information, as part of a formatted log entry as shown in Fig. 2. An exception may also be printed outside a formatted log entry, either because there is no corresponding handler, or because the handler chooses to do so.

Table 2 provides an overview of all the log entries contained in our studied bug reports, including the log level distribution, the total number of log entries that contain exceptions, and so on. Note that, most users do *not* upload the whole log in their bug reports. Instead, they often only upload log snippets (100 out of 106 bugs) that are likely related to the failure (e.g., close to the failure point).

2.5. Study method

All the findings in our study are based on our careful manual checking of the related materials of each bug, and the following steps are performed. a) Understand the failure symptoms by reading text descriptions and logs uploaded by users in bug reports. For example, the words “exiting”, “shutting down” and “stopped” usually appear in the logs when unexpected terminations occurs, and the words such as “crash”, “abort”, and “down” are often used to describe this type of failure. b) Figure out the causes of the bugs based on the bug-related log entries and the corresponding source code. Specifically, we started from the failure-symptom log entries and the source code where these log entries are printed, and traced back to find the execution path, referring to the before-failure-symptom log entries. c) Confirm the fixing strategies by checking the patches. Sometimes the patches may not directly fix the root causes. Thus, by reading the code changes in the final patch, we can clearly understand the logic for fixing. To guarantee what we did for debugging is consistent with developers’ ideas and the conclusions we made are correct, developers and users’ discussion for each bug is referenced during the above steps. Every bug and every finding is carefully examined by at least two authors who have previous experience with the studied distributed systems.

2.6. Threats to validity

Threats to the validity of our study come from several aspects.

Construct validity. One of the major contributions in this work is to quantify the extent to which the current system logs can help combat distributed bugs in practice. One threat is that we use bug reports to measure the effectiveness of logs in debugging. However, bug reports are the most effective and practical objects currently to conduct our study. Bug reports keep a good record of the bug life cycle, providing important basis for tracing back. Furthermore, we only look at bugs that are fixed and clearly described by users and developers. Particularly, we are concerned with the bugs whose bug reports contain logs. Consequently, those bugs that are not patched, or not described clearly, or do not have logs reported by users may have different characteristics from our observation.

Internal validity. The primary concern is about the correctness of our statistics. First, we have made our best efforts in conducting a comprehensive and unbiased study. To ensure the quality of the taxonomy in our study, each bug underwent several rounds of study by at least two people including categorization and examining source code.

Second, our statistics heavily depend on the information of bug reports that are reported by users and developers. The accuracy and the integrity of the provided information may impact our results. To limit this threat, we sampled the bugs whose information is relatively clear and complete. In addition, we have made efforts to verify and complete all the required information by checking the relevant source code.

Third, we did not reproduce the studied bugs, as with many previous empirical studies of bugs in large-scale systems [5,13,15–17]. In practice, it is actually obvious how developers diagnose — they either reproduce the bugs (based on the inputs and/or log) for diagnosis; or, they do not reproduce the bug and simply diagnose through inputs and/or logs. However, we are able to draw our conclusions based on the provided log snippets in bug reports. On the one hand, the information for the sampled bugs contained in the bug reports is sufficient. That is, the clues from logs, code, developers’ comments, patches, and others (such as thread dump and output of a certain command) are sufficient for us to understand the bug and the way to debug. On the other hand, the provided log snippets usually cover the symptoms of (and close to) the failure site (i.e., the starting point for debugging), sometimes as well as the suspicious log entries that are involved in the error propagation. Furthermore, based on the above clues and logs, we can follow the developers’ ideas and steps to figure out the other related log entries by checking the source code. Therefore, even if a complete log is provided, our statistics and conclusions might be slightly affected, and we believe that our work is still reasonable and valid. For example, with a complete log, a part of statistics in Table 2 may be changed, because the portion of failure-unrelated exceptions may be much larger. Note that figuring out the root cause does not necessarily need to know all steps to trigger the bug.

Table 3
Failure types of 106 distributed bugs.

Failure Type	# Bugs
Incorrect Results	37 (35%)
Hangs	26 (24%)
Unexpected Terminations	19 (18%)
Inconsistent States	13 (12%)
Resource Leaks/Exhaustions	6 (6%)
Data Losses	5 (5%)

External validity. The generalizability of our results is backed up by the following aspects. First, the target systems are representative distributed systems, and they also keep the diversity, including distributed computing, storage, and configuration and synchronization framework. Second, the bugs are not selected deliberately for a certain failure type concerning a specific system, and all the failure types of the studied bugs are consistent with the category in the existing work [14]. Third, all the bugs in our study have "distributed" attribute, and our study tried to explore some common patterns and features of them. Therefore, our findings and implications can also be applied to other distributed systems although it is not possible to generalize them across all distributed systems.

The statistics reported in our study may not be generalized to other datasets of distributed bugs, since the statistical results may vary with different bug sampling. And in our work, we sampled the bugs randomly so as to reduce the bias. Besides, given the different designs and implementations of different systems, it is very challenging to draw general conclusions in all distributed software projects. Thus we selected five widely used distributed systems, and all our findings are consistent among them.

Our study is specific to the logging mechanism used by our target systems — all the five target systems use Log4j for logging. In fact, Log4j is also used in Spark[9], Kafka[10] (a distributed streaming platform), and many other popular distributed systems. Of course, other logging libraries also exist. For example, the popular distributed messaging systems RabbitMQ[11] and ActiveMQ[12] use Lager[13] and SLF4J[14] for logging respectively. The popular in-memory data storage systems Memcached[15] and Redis[16] use their own built-in logging utilities. Fortunately, these different logging libraries all have similar fine-grained log levels as Log4j. Consequently, some of our conclusions could apply to systems using other logging libraries too. We plan to investigate this issue in other logging frameworks in future work.

3. Identifying failures

The starting point of failure diagnosis is clearly to identify the *existence* of a system failure and, if possible, identify the *type* of the failure. In this section, we study how helpful existing software logs are in helping these tasks. We determine the failure-type oracle by carefully checking the bug-report description and the source code, with the breakdown shown in Table 3. Specifically, we will answer the following research questions:

- RQ1.1: Is the failure occurrence of a distributed bug reflected by logs?
- RQ1.2: Can log analysis automatically tell the failure occurrence of a distributed bug?
- RQ1.3: Is the failure type of a distributed bug reflected by logs?
- RQ1.4: What log levels tend to reflect the failures of distributed bugs?
- RQ1.5: Are distributed bugs' symptoms distributed (in one node or multiple nodes)?

3.1. Is the failure occurrence of a distributed bug reflected by logs?

We check every bug in our bug set to see whether its failure symptom is reflected by software log entries and, if so, how, as shown in Table 4.

There are 8 bugs, whose failures are not correctly reflected by their logs ("NOT" column in Table 4). They fall into two categories.

For 5 bugs, the information conveyed by logs is wrong or misleading. That is, the log reports that some actually healthy system components do not work normally — what actually fails is the component that prints these logs. Take hd1371 as an example, DFSClient printed that "Could not obtain block" due to IOException "No live nodes contain current block" for many data blocks. It turned out that these data blocks were all healthy and what went wrong was the DFSClient who generated these logs.

For three bugs, their failures happen silently and did not leave any log symptoms. For example, in zk1382, users did not notice memory leaks, until they ran a special debugging command `wchp` and found out that the system was keeping some watches and related data structures that belonged to expired sessions. However, no warnings or any leak related information could be found in logs.

Other bugs' failures can all be correctly identified by log entries. For more than half (56%) of the bugs, their failures are reflected by exceptions in the log, which matches the previous empirical findings that many catastrophic failures are resulted from incorrect error handling [14]. Long sequences of repeated log entries, and inconsistency among log entries are also common log symptoms, as shown in Table 4. Note that for 5 bugs, the log symptom is described in natural language in the log, although no exceptions were thrown. For example, in hb8519, when the backup HMaster was trying to take over the whole system, it stopped silently, making the entire system unavailable. Although no exception was thrown, the log did contain an INFO entry, saying "*Cluster went down before this master became active*".

Finally, as shown in Table 5, the trend discussed above is consistent across software projects: exceptions in log entries and repeated log entries are the two most common log-symptom patterns in all the five applications studied by us.

Summary For most (93%) distributed bugs, the occurrences of their failures are reflected by log symptoms, such as exceptions, repeated log entries, and inconsistent log entries (88 bugs fall into these three most common patterns). However, there are bugs (7%), whose failures cannot be correctly identified by software logs either due to misleading log entries, or the lack of log information, and impose challenges to failure diagnosis.

Finding 1. For about 20% (15 out of 88) of the distributed bugs whose failures are reflected by three common log patterns (exceptions, repeated log entries, and inconsistent log entries), logs from more than one node are required to determine the failure occurrences.

Implication. Although exceptions and repeated log entries are also common for bugs in single-machine systems, they will take on a different meaning in distributed bugs by gathering logs from different nodes. In the settings of distributed systems, the seemingly abnormal logs in a single node could be some intermediate errors which are finally tolerated, while the seemingly trivial logs in a single node could be fatal.

3.2. Can log analysis automatically tell the failure occurrence of a distributed bug?

Since we can manually tell most bugs' failure occurrences from logs, naturally we want to know whether an automatic log analysis tool is

[9] https://spark.apache.org.
[10] https://kafka.apache.org.
[11] https://www.rabbitmq.com.
[12] http://activemq.apache.org.
[13] https://github.com/erlang-lager/lager.
[14] https://www.slf4j.org.
[15] https://memcached.org.
[16] https://redis.io.

Table 4
How different types of failures are reflected by log entries (short as *entries* in table).

Failure Type\Log	Exceptions in entries	Repeated entries	Inconsistent entries[a]	Log text	Suspended entries[b]	NOT
Incorrect Results	35	0	0	2	0	0
Hangs	1	20	0	0	5	0
Unexpected Terminations	18	0	0	1	0	0
Inconsistent States	0	0	6	2	0	5
Resource Leaks/Exhaustions	2	1	0	0	0	3
Data Losses	3	1	1	0	0	0
Total (%)	59 (56%)	22 (21%)	7 (6%)	5 (5%)	5 (5%)	8 (7%)

[a] Log entries that reflect the inconsistent states of the same object.
[b] The program does not complete, but at some point, the following log entries (associated with a certain task) that should be output in a successful run will not print.

Table 5
How failures in different applications are indicated by log entries.

App.\Log	Exceptions in entries	Repeated entries	Inconsistent entries	Log text	Suspended entries	NOT
MapReduce	14	5	2	0	2	3
HDFS	12	1	0	0	0	2
HBase	11	9	2	2	0	0
Cassandra	20	3	3	3	3	1
ZooKeeper	2	4	0	0	0	2

$$\left[\begin{array}{l} LE\ 1 \\ \quad\dots \\ LE\ n \end{array}\right. \qquad \left[\begin{array}{l} LE\ 1\ (id) \\ \quad *** \\ LE\ 2\ (id) \\ \quad *** \\ \quad\dots \\ LE\ n\ (id) \end{array}\right. \qquad \langle LE\ (id1, id2, \dots, idn)\rangle$$

(i) (ii) (iii)

Fig. 3. Three patterns of repeated log-entry (short as *LE*) symptom. (The left bracket contains the minimum repetitive log-entry units.).

able to do that. Our study shows that, although promising, there are non-trivial challenges.

An automated tool can easily identify log entries associated with exceptions. However, the challenge is that many exceptions in the logs are **not** log symptoms. As shown in the last three columns of Table 2, among all exceptions contained in log snippets that are provided by user bug reports, ***about one third*** of them are **not** log symptoms. Some of them are completely unrelated to the bug, and some are actually the bug triggers (e.g., one node fails to connect with another node due to transient network failures), instead of the log symptoms. Note that, most bug reports only contain log snippets that are considered useful for debugging by users (e.g., recent logs). If we look at the whole logs, the portion of failure-unrelated exceptions would be much larger. Fortunately, we find that for 44% (26 out of 59) of the distributed bugs whose failures are reflected by exception log symptoms, the exception-related logs can be used to determine the failure automatically. And these exceptions fall into *two types*: exceptions that lead to unexpected termination, and null pointer exceptions. Note, although null pointer exceptions can also lead to program termination, they will not trigger generic termination methods such as `abort`, `exit`, and `stop` as unexpected termination. Thus, we list them separately.

A carefully designed log analysis tool should be able to identify repeated log-entry symptoms, particularly since, in all the 22 cases where log symptoms are repeated log entries, the repetition is infinite. Fig. 3 indicates *three patterns* of log-entry repetition observed in our study. The most common pattern repeats a sequence of adjacent entries infinitely (**(i)** in the figure); sometimes, the repeated log entries are not adjacent, but they all contain the same ID (**(ii)** in the figure); sometimes, log entries that follow the same template but are instantiated with different IDs are generated infinitely (**(iii)** in the figure). These three patterns could appear on one or multiple nodes, and could be mixed with each other.

Failures could be reflected by inconsistent state descriptions of the same object among adjacent log entries in one node (e.g., Fig. 4) or log entries generated at similar time by multiple nodes (e.g., Fig. 5). To automatically identify such inconsistencies, we need to *identify* related log entries based on object IDs and timestamps within one node or across nodes. We also need to *differentiate* regular object-state changes (e.g., a job status changes from INITED to RUNNING) from irregular inconsistencies, which can benefit from system state-machine knowledge.

Some natural language processing techniques can help automatically tell the occurrence of failures indicated by log text descriptions. Finally, suspended log-entry symptoms are all related to hang bugs in our study. Take ca5804 as an example, a Cassandra node fails to send tree requests to the other peer nodes during a repair session. Consequently, the repair cannot proceed and hangs silently, due to the responses which will never come. Some *simple* strategies can help detect this. For example, the log entry sequence of a certain successful transaction can be collected through the specified id (such as the repair session id in ca5804), then we can monitor the representative log entries in stages by setting a time limit.

Summary Apart from 33 bugs with exception log symptoms, for 66% (65 out of 98) of the distributed bugs whose failures are reflected by logs, failure occurrences can be automatically analyzed. To automatically identify the failures, the insights and strategies include two types of exceptions that can be used as failure indicators, three patterns of repeated log entries, two requirements for identifying inconsistent log entries, and the simple approaches for bugs with log-text and suspended-entry symptoms, as discussed above. Automatically identifying failure occurrences from logs can save users' manual debugging efforts and help start failure recovery early, preventing further damages made by the failed system. Our study points out both opportunities and challenges in conducting such automatic log analysis.

Finding 2. About one third of the exceptions provided in bug reports are not failures' log symptoms.

Implication. Exceptions contained in bug reports play different roles. They could be log symptoms, bug triggers or completely unrelated to the bug. The unrelated exceptions are noises and they can mislead the log analysis tools to infer an unrelated or incorrect execution path. When designing or using automated log analysis tools, these unrelated exceptions should be carefully considered and filtered out. Meanwhile, the

```
(path = org.apache.hadoop)
2012-09-27 20:28:38,068 [main] INFO path.mapred.ClientServiceDelegate - Application state is completed.
FinalApplicationStatus=SUCCEEDED. Redirecting to job history server
2012-09-27 20:28:38,530 [main] WARN path.mapred.ClientServiceDelegate - Error from remote end:
Unknown job job_1348097917603_3019
```

Fig. 4. Inconsistent log entries from mr4691: a job is announced as completed then immediately as unknown.

```
node1 (127.0.0.3)
① INFO [StorageServiceShutdownHook] 2013-02-14 10:01:11,115 Gossiper.java (line 1134) Announcing
shutdown
node2
② INFO [GossipStage:1] 2013-02-14 10:01:12,119 Gossiper.java (line 831) InetAddress /127.0.0.3 is now
dead.
node3
② INFO [GossipStage:1] 2013-02-14 10:01:12,119 Gossiper.java (line 831) InetAddress /127.0.0.3 is now
dead.
③ INFO [GossipStage:1] 2013-02-14 10:01:12,238 Gossiper.java (line 817) InetAddress /127.0.0.3 is now
UP
```

Fig. 5. Inconsistent log entries from ca5254: after node1 shuts down, it is incorrectly considered UP by node3 shortly after being considered dead.

Table 6
Resource leaks and data losses bugs whose failure types cannot be determined by the log alone (Bugs whose log symptoms are from multiple nodes are underlined).

Failure Type\Log	Exceptions in entries	Repeated entries	Inconsistent entries	Log text	Suspended entries
Resource Leaks/Exhaustions	1	<u>1</u>	0	0	0
Data Losses	3	0	<u>1</u>	0	0

same exception will also play different roles in different bugs or contexts. Typically, the common connection exceptions are harmless when they are successfully tolerated by reconnecting, but if the destination node itself is invalid, the source node should be rescued from the continuous connection exceptions caused by the nonstop reconnection.

3.3. Is the failure type of a distributed bug reflected by logs?

For those bugs whose failure occurrences can be identified from logs, it is further expected that the exact failure type can also be identified from the logs.

As shown in Table 4, some failure types are easy to decide based on logs. For example, almost all repeated-entry symptoms reflect hangs and similarly most hangs are reflected by repeated entries; almost all inconsistent-entry symptoms reflect inconsistent-state failures and similarly most inconsistent-state failures are reflected by inconsistent-entry symptoms.

Both incorrect results and unexpected node/job terminations are all/mostly reflected by exceptions in the log. However, it is not trivial to tell which failure type an exception belongs to. If one only looks at logs, the log level is the best generic classifier that we can find: the failure type is much more likely to be an unexpected termination, given a FATAL log symptom entry (11 out of 13 cases); the failure type is much more likely to be an incorrect result, given an ERROR log symptom entry (24 out of 43 cases). Source code is actually a more accurate source to tell these two types apart — unexpected terminations are always triggered by generic termination methods like `abort`, `exit`, or `stop`. It would be nice if the corresponding logs could also be standardized.

For the last two failure types in Table 4 (i.e., resource leaks/exhaustions and data losses), often, the exact failure type cannot be easily decided by examining the logs only, as shown in Table 6. For example, in hb3722, the ERROR entry from HMaster complains about failing to split HLog for the crashed region server due to ConnectException. Without checking the source code, users (unless they are HBase experts) have no way to realize that HMaster will skip this failed splitting, which eventually leads to data losses.

Table 7
How different types of failures are indicated by log levels.

Failure Type\Log Level	FATAL	ERROR	WARN	INFO	No level
Incorrect Results	0	24	7	3	3
Hangs	1	9	6	10	0
Unexpected Terminations	11	5	0	1	2
Inconsistent States	0	2	3	3	0
Resource Leaks/Exhaustions	0	2	0	1	0
Data Losses	1	1	1	2	0
Total	13	43	17	20	5

Summary Whether the failure type can be reflected by the logs closely relates to the failure type. Hangs and inconsistent states are easy to tell based on their log symptoms. Unexpected terminations and incorrect results can often be decided by the logs, and occasionally require some light-weight code checking. For the above failure types, future work can automatically analyze logs and some code regions localized by specific log entries to report failure types. However, for silent failures, including resource leaks/exhaustions and data losses, more log information or advanced log/code analysis techniques are expected to help determine them and help further debugging.

3.4. What log levels tend to reflect the failures of distributed bugs?

Log levels (e.g., FATAL, ERROR, WARN and INFO) are meant to indicate different significances of log entries and help developers prioritize their debugging efforts. Therefore, we check all the 98 bugs whose failures are reflected in logs to see what are their log symptom levels. The results are shown in Table 7 (if a bug has multiple symptom log entries with different levels, we count the highest level).

Originally we thought most failures are reflected by FATAL or ERROR log entries, but this is **not** true based on our study. For ***only about half*** (53%) of the distributed bugs, the failure occurrences are reflected by FATAL or ERROR log entries.

FATAL entries As shown in Table 7, FATAL log entries appear in 13 bugs' symptoms. They are highly correlated with unexpected termina-

```
2011-07-21 07:05:55,835 FATAL org.apache.hadoop.hbase.master.HMaster: Error deleting OPENED node
in ZK for transition ZK node...
2011-07-21 07:08:03,288 INFO org.apache.hadoop.hbase.master.ActiveMasterManager: Another master is
the active master, 158-1-91-103:20000; waiting to become the next active master
```

Fig. 6. Log snippet from hb4124.

Table 8
Log symptom patterns for different log levels.

Log\Log Level	FATAL	ERROR	WARN	INFO
Exceptions in entries	12	32	6	4
Repeated entries	1	9	7	5
Inconsistent entries	0	1	2	4
Log Text	0	1	2	2
Suspended entries	0	0	0	5
Total	13	43	17	20

Table 9
How failures in different applications are indicated by log levels.

App.\Log Level	FATAL	ERROR	WARN	INFO	No level
MapReduce	2	10	1	7	3
HDFS	2	4	5	2	0
HBase	8	5	6	4	1
Cassandra	0	22	3	6	1
ZooKeeper	1	2	2	1	0

tions that are indeed often severe — 9 bugs with FATAL log symptoms led to master-node crashes and then the whole system downtime.

However, not all FATAL log entries indicate failures. There are two bugs in our bug set whose logs contain FATAL entries nearby the eventual log symptoms but are **not** log symptoms. For example, in hb4124, although a HMaster crashes, reflected by a FATAL entry in Fig. 6, this crash is not the bug symptom. Instead, the bug occurs during the interaction between the new active HMaster and a region server.

ERROR entries ERROR log entries appear in 44% of all log symptoms and are the most common log symptom level. It covers all failure types, with incorrect results being the most related failures.

WARN, INFO, and others Surprisingly, about 40% of the bugs' log symptoms have levels **lower** than ERROR, and they cover **all** failure types. This is due to several reasons, based on our study in Table 8.

First, although each individual WARN or INFO entry may be insignificant, they reflect failures (e.g., hangs) as part of infinite log repetitions. This contributes to 12 bugs whose symptoms are WARN (7 bugs) or INFO (5 bugs) entries.

Second, some WARN or INFO entries are associated with exceptions that developers (incorrectly) deemed not severe (10 bugs). For example, in hd4233, HDFS developers initially considered an IOException "Unable to start log segment" as insignificant. They associated this exception with merely an INFO log level and let NameNode continue its serving. However, based on users' report about data loss, developers realized that they should have "*taken a corrective action or regard this condition as FATAL*".

Third, only when looking at the WARN or INFO entries from a global view (by correlating the low-level log entries from different nodes), will the latent problems be exposed, such as the INFO entries we discuss in ca5254 (Fig. 5). This is also unique in distributed bugs.

Finally, five bugs' log symptoms are unformatted exception log entries that are not associated with any log level.

Across software projects Table 9 shows log symptoms' log level distribution in different software projects. As we can see, the trend is consistent across different projects.

Summary Almost half of the bugs' symptoms are reflected by low-level log entries. Meanwhile, not all high-level log entries reflect real system failures. The latent results and potential influences of low-level entries should be considered, since low-level log entries that follow particular patterns or come from multiple nodes can uncover the real problems.

Finding 3. For only about half (53%) of the distributed bugs, the failure occurrences are reflected by FATAL or ERROR log entries.

Finding 4. FATAL are not always fatal, and INFO could be fatal.

Implication. The severity of a certain log entry could be inconsistent with the pre-defined log level. This is because the severity of a log entry could change with different contexts, when we consider the logs before/after the current log entry, the logs from a different node about the same time, or the current log appears in a special pattern (e.g. repeating). The mismatch between log level and the actual severity of the log entry also exists in single-machine system bugs but occurs more often in distributed bugs and the context becomes more complicated than that in single-machine systems, due to the cross-node dependency. For example, the redundancy mechanism in distributed systems not only exists inside nodes but also exists across nodes, and the states of the other nodes can change the implication and severity of the log entry in the current node.

3.5. Are distributed bugs' symptoms distributed?

Intuitively, a distributed bug's symptom could be *local* (i.e., from a single node) or *global* (i.e., from multiple nodes).

14% of our studied distributed bugs have global symptoms, as shown in Fig. 7. They cover all failure types except unexpected terminations. These global log symptoms fall into two types: (1) globally inconsistent log entries — logs are consistent if looking at each node's log individually, but are inconsistent if comparing different nodes' logs together (e.g., ca5254 in Fig. 5); (2) globally repeated log entries, following three patterns, as shown in Fig. 8.

(i) Repeated log entries indicating state S appear in Node-1, and if the log entry indicating state S' appears in Node-2, the repeated entries on Node-1 will be infinite and the failure exists.
(ii) Similar with (1), except that state S' in Node-2 also repeats.
(iii) Two or more homogeneous nodes are involved. Each node is stuck in the same state S indicated by the repeated log entries, but state S is not the steady state a system should be.

Summary Some distributed bugs' symptoms are scattered around multiple nodes, and they are easily ignored. Our study reveals several global symptom patterns that only apply to distributed bugs, and these patterns can be used to develop automated log analysis tools, for helping system administrators discover the failures easier.

Finding 5. The failures of a non-negligible part (14%) of distributed bugs are reflected by global log symptoms.

Implication. The trivial log entries that are assigned low log levels, or do not follow any special pattern, will become non-trivial when they are correlated and considered in a global view. Identifying failures of these bugs is challenging and can benefit from automated log analysis tools. Global symptom is unique to distributed bugs due to their distributed nature: the cooperation and distributed states across nodes.

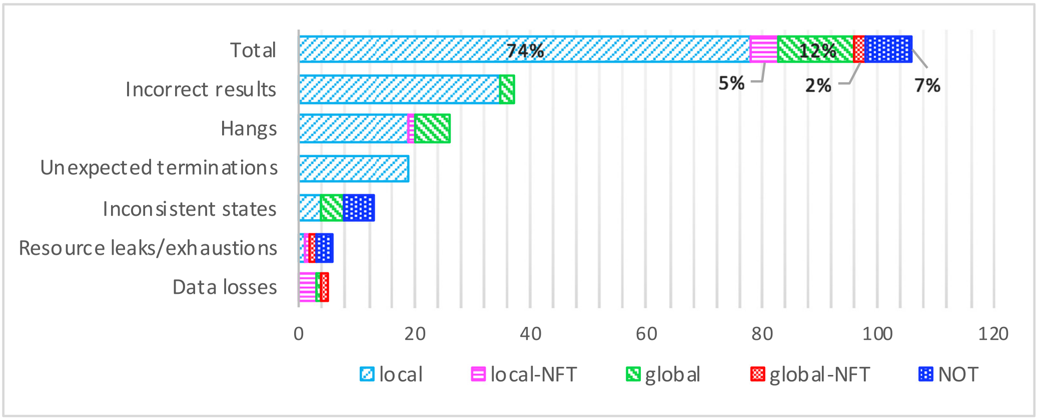


Fig. 7. Distribution of log symptoms for different failure types (NFT: log symptom does Not reflect Failure Type; NOT: 8 bugs whose failures are not reflected by their logs.).

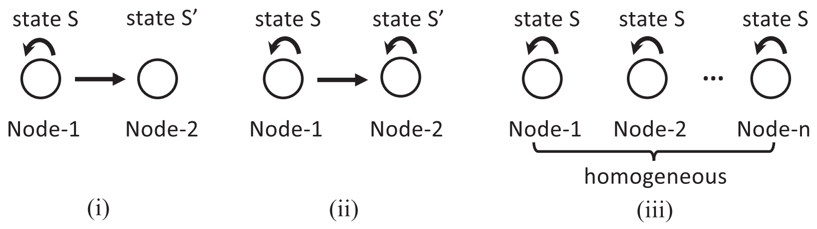
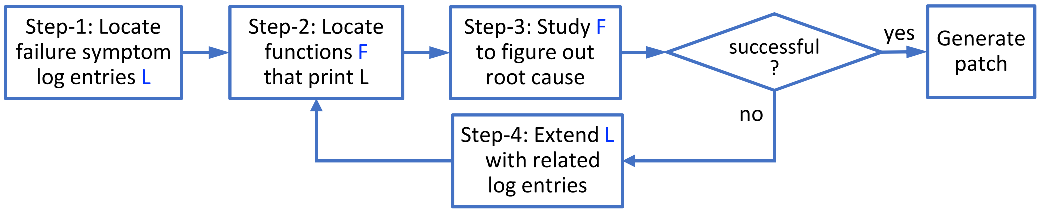


Fig. 8. Three patterns of repeated-log-entry-related global symptoms.



Step-1: Locate failure symptom log entries L
Step-2: Locate functions F that print L
Step-3: Study F to figure out root cause
successful ?
yes
Generate patch
no
Step-4: Extend L with related log entries

Fig. 9. A general flow of debugging distributed bugs (Step-2 is straightforward when L is associated with exception stack traces. Otherwise, it is done by searching source code using keywords in the log messages.).

Table 10
Root-cause types of 106 distributed bugs.

Root-Cause Type	# Bugs
CODE	37
CODE + VALUE	36
CODE + TIME	10
CODE + VALUE + TIME	23

Table 11
Items needed for root-cause diagnosis.

Needed items	# Bugs
(a) Log symptom entries alone	38 (35%)
(b) Log symptom and other bug-related entries	57 (54%)
(c) Log symptom entries and thread dump	5 (5%)
(d) Other	6 (6%)

4. Diagnosing failure root causes

Identifying the occurrence of a failure and the failure type is only the starting point of failure diagnosis. Next, developers need to work on locating the root cause of the failure. There is no standard procedure for debugging, since the debugging expertise and habits of developers ranging from novices to experts vary a lot [23]. Based on our study on the target bug reports, a general flow of debugging distributed bugs is described in Fig. 9.

The root cause of the failure could be a specific set of code (denoted by CODE in Table 10), or certain values of specific program variables (denoted by VALUE in Table 10), or certain timing among multiple threads or nodes (denoted by TIME in Table 10), or a combination of them. To better understand the diagnosis process for real-world distributed bugs, we first manually identify the root cause of every bug based on bug-report description, patch, and the source code. And the result of root-cause distribution is shown in Table 10. Then we aim to answer the following research questions:

- RQ2.1: Are log entries sufficient for root-cause diagnosis?
- RQ2.2: How to find bug-related log entries (in addition to symptom entries) for debugging? Particularly, are they all inside the symptom node(s) or are they distributed?
- RQ2.3: How to figure out root causes based on bug-related log entries?

4.1. *Are log entries sufficient for root-cause diagnosis?*

Table 11 shows what information is needed to figure out a bug's root cause. We consider a bug to be successfully diagnosed by a certain set of log entries, if the information contained in these entries, including timestamps, variable values, log-statement location in source code, exception stack trace, and others, is sufficient to help identify the root-

Table 12
How to locate non-symptom log entries for diagnosis.

Node	Inside Symptom Nodes			Outside Symptom Nodes		Total
	ID	Adjacent	Both	Timestamp+	Timestamp	
Local	9	7	5	-	-	21
Global	-	-	-	20	2	22
Local+Global	1	10	3	14	0	14

* The "Node" column indicates whether the log entries to be located are in symptom nodes (Local) or non-symptom nodes (Global) or both.

cause component of the bug (i.e., CODE, VALUE, and/or TIME). Our judgment is based on both our own manual diagnosis experience and also developers' experience described in bug reports.

Table 11 shows that almost 90% of the bugs can be troubleshot based on existing log entries (categories (a) and (b) in Table 11). There are 11 bugs whose root causes cannot be diagnosed based on logs alone. Five of them are hang bugs, and developers could not figure out their root causes until users provided thread dumps. The other 6 bugs are all timing-related bugs (i.e., CODE + VALUE+ TIME root cause), the manifestation of which requires fine-grained special timing that is not reflected by any information in the log. For example, in mr4691, developers explained that the root cause is "a race condition in the historyserver where *two threads* can be trying to scan the same user's intermediate directory for two separate jobs. One thread will win the race and update the user timestamp in `HistoryFileManager.scanIntermediateDirectory` before it has actually completed the scan. The second thread will see the timestamp updated, (mistakenly) think there is no point in doing a scan, and return with no job found." Unfortunately, no log entries are printed by these two threads nearby the racing operations, making diagnosis difficult.

However, Table 11 also shows that log-symptom entries alone are only able to diagnose slightly more than one third of the bugs. For more than half of the bugs, their root causes can be diagnosed based on log entries, but extra work is needed to identify the log entries that are not log symptoms.

Summary Log information is sufficient to help diagnose close to 90% of the bugs. However, it is **non-trivial** to identify all the useful log entries. For more than half of the bugs, diagnosis requires log entries that are not part of the bug symptoms. Log support for diagnosing hang bugs and timing bugs still needs improvement.

4.2. How to find bug-related log entries that are not failure symptoms?

As discussed above, root-cause diagnosis often relies on log entries that are not part of the log symptoms. Therefore, we study how to locate these log entries below.

Are they all inside the symptom node(s)? We first check whether such non-symptom bug-related log entries for a bug are all inside the node(s) where log symptom entries are located, denoted by "Local" in the "Node" column of Table 12, or outside the symptom node(s) ("Global" in Table 12), or both. As shown by Table 12, non-symptom nodes' logs need to be checked in *almost two thirds* of the cases (36 out of 57 bugs), reflecting the truly *distributed* nature of these bugs, which is unique compared to debugging single-machine software.

How to locate bug-related log entries in symptom nodes? As shown in Table 12, when searching log entries inside the symptom nodes, one can often leverage the IDs included in the log symptom entries to locate the related log entries that are useful for debugging (column "ID" in Table 12), or one can simply check backward for entries recorded right before the symptom entries (column "Adjacent" in Table 12). The before-symptom log entries are quite important, especially when it is hard to reproduce the bug. These entries are also the diagnosis basis for developers in practice. Such as in ca5105, the assignee of this bug suggested that "I'm trying to reproduce myself but still no luck. Some logs before error happened may help if you can upload."

How to locate bug-related log entries in non-symptom nodes? When searching log entries beyond the symptom nodes, the timestamps recorded in the log symptom entries are very useful for locating related entries on non-symptom nodes. However, our study shows that merely depending on timestamps is usually **not** sufficient. In 34 out of 36 cases (denoted by "Timestamp + " in Table 12), one needs to first identify the bug-related non-symptom node(s) and then apply the timestamp(s) to identify the related log entries on that/those node(s).

There are mainly two ways to identify bug-related non-symptom nodes: (1) leveraging IDs in symptom log entries and (2) leveraging RPC exceptions in symptom log entries. For the former, the symptom entries sometimes contain the IP address or the symbolic ID of another node. For example, in hd5671 (Fig. 10), we get an INFO-level symptom log entry on DFSClient, showing that some token-related errors occur when connecting to DataNode. Using the timestamp of this INFO, the block ID "-8825057745517l3967" and the token error, we successfully locate the ERROR entry on DataNode. The exception stack trace contained in this ERROR record helps us understand how the exception is generated. For the latter, some symptom entries contain RPC exception returned by a remote node that executed the RPC. Tracking log entries on that remote node is usually helpful for root-cause diagnosis. For example, in hb3617, the exception stack trace from HMaster log (Fig. 11) clearly shows that the exception occurs when HMaster sends CLOSE RPC to a certain region server node. We can then check the logs of that region server node.

The two ways discussed above are also reflected by developers' discussion. Take mr2998 as an example, the bug reporter complained that he was getting an exception frequently when running his jobs and only posted the logs from JobClient (Fig. 12). The exception stack trace clearly shows that this happens when JobClient tries to contact AM for job progress through RPC. Then the developer commented "From both the logs you pasted, the job clearly failed, you need to debug why by looking at AM logs." Later on, the reporter found the WARN[17] log entry from NodeManager (NM) which is hosting the AM (running in the container whose ID ends with "000001"). And this warning message reveals that AM is being killed because of the abnormal memory consumption. That is why JobClient fails to contact the AM. Then the developers continue to figure out why the AM was using so much memory.

Summary IDs and the before-symptom log entries are the basis of identifying bug-related log entries inside the log symptom nodes. What is more, identifying bug-related log entries often requires going beyond the log symptom nodes (*more than one third* of all bugs). Particularly, IDs and RPC exceptions in symptom log entries are the two primary ways for identifying the non-symptom nodes.

Finding 6. For more than half (54%) of the distributed bugs, diagnosis requires log entries that are not part of the log symptoms. Particularly, for about two thirds of these bugs, bug-related log entries go beyond the log symptom nodes.

Implication. In the scenario of distributed bugs, different nodes often need to communicate and work cooperatively to accomplish a task deferring to the predefined distributed system protocols. This is the insight under which we are able to locate non-symptom nodes mainly through timestamps (event order across nodes), IDs (the same object operated across nodes) and RPC (communication across nodes). There is no single routine in identifying these log entries locally or globally (i.e., beyond one node). Future research can work on helping developers identify these debugging-needed nodes and log entries based on symptom

[17] The log entries on JobClient and NM posted in the bug report do not come from the same job. But actually for the same job, the WARN on NM should happen before the INFO on JobClient, and the job ID (part of the container ID) included in the WARN should be consistent with the job ID in the INFO.

```
(path = org.apache.hadoop)
DFSClient
2013-12-13 15:48:31,474 INFO path.hdfs.DFSClient: Will fetch a new access token and retry, access token
was invalid when connecting to /192.168.2.27:1004: path.hdfs.security.token.block.InvalidBlockTokenExce-
ption: Got access token error for OP_READ_BLOCK, self=/192.168.2.27:56975, remote=/192.168.2.27:
1004, for file /hbase/XXXX/b50bf1b95c9242cdd242dc4e6549bc90/raw/d59819ebe5574c79a5d1cf13a733d2e
d, for pool BP-621472495-192.168.2.25-1375176775166 block -8825057745517139677_11426277
DataNode
2013-12-13 15:48:31,474 ERROR path.hdfs.server.datanode.DataNode: dn2:1004: DataXceiver error
processing READ_BLOCK operation src: /192.168.2.27:56975 dest: /192.168.2.27:1004
path.security.token.SecretManager$InvalidToken: Block token with block_token_identifier (...blockPoolId=
BP-621472495-192.168.2.25-1375176775166, blockId=-8825057745517139677...) is expired.
... (stack trace)
```

Fig. 10. Log snippet from hd5671.

```
(path = org.apache.hadoop.hbase)
2011-03-10 07:48:39,192 FATAL path.master.HMaster: Remote unexpected exception
java.net.NoRouteToHostException: No route to host
... (stack trace)
at path.ipc.HBaseClient.getConnection(HBaseClient.java:883)
at path.ipc.HBaseClient.call(HBaseClient.java:750)
at path.ipc.HBaseRPC$Invoker.invoke(HBaseRPC.java:257)
at $Proxy6.closeRegion(Unknown Source)
at path.master.ServerManager.sendRegionClose(ServerManager.java:589)
at path.master.AssignmentManager.unassign(AssignmentManager.java:1093)
... (stack trace)
at path.master.HMaster.balance(HMaster.java:692)
at path.master.HMaster$1.chore(HMaster.java:583)
at path.Chore.run(Chore.java:66)
2011-03-10 07:48:39,192 INFO path.master.HMaster: Aborting
```

Fig. 11. Log snippet from hb3617.

```
(path = org.apache.hadoop)
JobClient
11/09/12 17:18:01 INFO mapred.ClientServiceDelegate: Failed to contact AM/History for job
job_1315847180566_0006 Will retry..
java.lang.reflect.UndeclaredThrowableException
at path.mapreduce.v2.api.impl.pb.client.MRClientProtocolPBClientImpl.getTaskAttemptCompletionEve-
nts(MRClientProtocolPBClientImpl.java:179)
... (stack trace)
at java.lang.reflect.Method.invoke(Method.java:597)
at path.mapred.ClientServiceDelegate.invoke(ClientServiceDelegate.java:237)
at path.mapred.ClientServiceDelegate.getTaskCompletionEvents(ClientServiceDelegate.java:276)
... (stack trace)
NodeManager (hosting the AM)
11/09/15 12:46:51 WARN monitor.ContainersMonitorImpl: Container [pid=22125, containerID=container
_1316090748961_0001_01_000001] is running beyond memory-limits. Current usage: 2150408192bytes.
Limit : 2147483648 bytes. Killing container.
```

Fig. 12. Log snippet from mr2998.

log entries, by analyzing local and global timestamps, exceptions, and ID information recorded in logs.

4.3. How to figure out root causes based on bug-related log entries?

With all the bug-related log entries identified, finally, one needs to work on figuring out the root cause. Since this is a very complicated and hard-to-quantify procedure, we only highlight some of the high-level trends that we have experienced below.

First, we find exception stack traces recorded in logs very useful in identifying root causes. In fact, they help *more than two thirds* of the bugs in our study (74 bugs in total). Particularly, 9 bugs can be troubleshot based on stack trace alone in log symptom entries. Take hb3617 (Fig. 11) as an example, the stack trace clearly shows that NoRoutToHostException, a network socket error, which occurs during RPC `closeRegion` has made the HMaster node abort. The stack trace also tells us this RPC is invoked as part of the periodic region-balancing job `balance`. Based on this stack trace, developers quickly figured out the root cause — an over-reaction (i.e., `abort` the master node) to an inevitable network error during a non-critical periodic job.

Of course, stack trace alone is not sufficient for diagnosing most bugs. We find ourselves and developers working hard to re-establish bug-related control flow and data flow relationship to figure out root causes, just like how non-distributed bugs are diagnosed [11]. Here we make an illustration with hb8230. The log symptom of this bug is the unexpected NullPointerException (NPE) during RegionServer (RS)'s abort, as show in Fig. 13. Based on the line number from the top frame on NPE stack trace and the source code, we know NPE is ini-

Table 13
Different strategies for fixing root causes.

Strategies for fixing root causes		# Bugs
(a) fixing timing problems	add synchronization (11)	19
	fast-forward or postpone specific operation (7)	
	add new checkpoint (1)	
(b) fixing semantics problems	add sanity check (17)	52
	change ID format (4)	
	others (31)	

```
(path = org.apache.hadoop.hbase, path1 = org.apache.hadoop.hdfs)
Exception in thread "regionserver26003" java.lang.NullPointerException
at path.replication.regionserver.Replication.join(Replication.java:129)
at path.replication.regionserver.Replication.stopReplicationService(Replication.java:120)
at path.regionserver.HRegionServer.join(HRegionServer.java:1803)
at path.regionserver.HRegionServer.run(HRegionServer.java:834)
at java.lang.Thread.run(Thread.java:662)

Before NPE occurs
2013-03-29 10:32:42,260 FATAL [regionserver26003] ABORTING region server om-host2, 26003,
1364524173470: Unhandled exception: cannot get log writer path.regionserver.HRegionServer.abort(HRe-
gionServer.java:1737)
java.io.IOException: cannot get log writer
at path.regionserver.wal.HLog.createWriter(HLog.java:757)
... (stack trace)
at path.regionserver.HRegionServer.instantiateHLog(HRegionServer.java:1327)
at path.regionserver.HRegionServer.setupWALAndReplication(HRegionServer.java:1316)
at path.regionserver.HRegionServer.handleReportForDutyResponse(HRegionServer.java:1030)
at path.regionserver.HRegionServer.run(HRegionServer.java:706)
at java.lang.Thread.run(Thread.java:662)
Caused by: java.io.IOException: path1.server.namenode.SafeModeException: Cannot create file
/hbase/.logs/om-host2,26003,1364524173470/om-host2%2C26003%2C1364524173470.1364524361366.
Name node is in safe mode. The reported blocks 14 has reached the threshold 0.9990 of total blocks 14.
Safe mode will be turned off automatically in 21 seconds.
at path1.server.namenode.FSNamesystem.startFileInternal(FSNamesystem.java:1601)
at path1.server.namenode.FSNamesystem.startFile(FSNamesystem.java:1547)
at path1.server.namenode.NameNodeRpcServer.create(NameNodeRpcServer.java:412)
... (stack trace)
at path.regionserver.wal.HLog.createWriter(HLog.java:754)
... 10 more
```

Fig. 13. Log snippet from hb8230.

tially triggered on the variable *replicationSink* (Fig. 14). Then we need to figure out why this variable is null. After checking its data flow, we find that *replicationSink* should be instantiated during the invocation of `startServiceThreads`. Meanwhile, based on the stack trace of SafeModeException, we get another part of control flow (marked in blue in Fig. 14), showing that due to SafeModeException triggered in NameNode, `handleReportForDutyResponse` throws IOException and `abort` is invoked. Consequently, `startServiceThreads` will not be executed, and *replicationSink* will not be instantiated, leading to the null dereference. Therefore, one of the developers comments “any exception occurs before `startServiceThreads` may cause this NPE” and “what caused the log writer creation failure’ is not the key point”. Furthermore, when stack traces are not available, one can locate log entries in source code and use that to bootstrap failure diagnosis.

Summary Once bug-related nodes and log entries are identified, stack traces or the string constants of the log entries may be utilized to build (part of) the bug-related control flow. In addition, the variable which directly or indirectly results in the failure may bootstrap the data flow checking, and complement the already built control flow or data flow.

5. Fixing bugs

The ultimate goal of failure diagnosis is to produce patches. In this section, we carefully study bug patches to understand the following research questions:

- RQ3.1: How are distributed bugs fixed?
- RQ3.2: Are logs changed in bug patches?
- RQ3.3: What is the relationship between logs and patches?
- RQ3.4: How quickly are distributed bugs fixed?

5.1. How are distributed bugs fixed?

At a high level, we categorize all patches into two types: *fixing root causes* and *fixing end symptoms*.

Our study finds 67% (71 out of 106) of the bugs are patched by fixing root causes — those intermediate errors and failure-inducing exceptions caused by the original bugs are all eliminated. As shown in Table 13, the approaches taken by these patches are similar to traditional non-distributed bugs. About 30% of error-fixing patches tackle

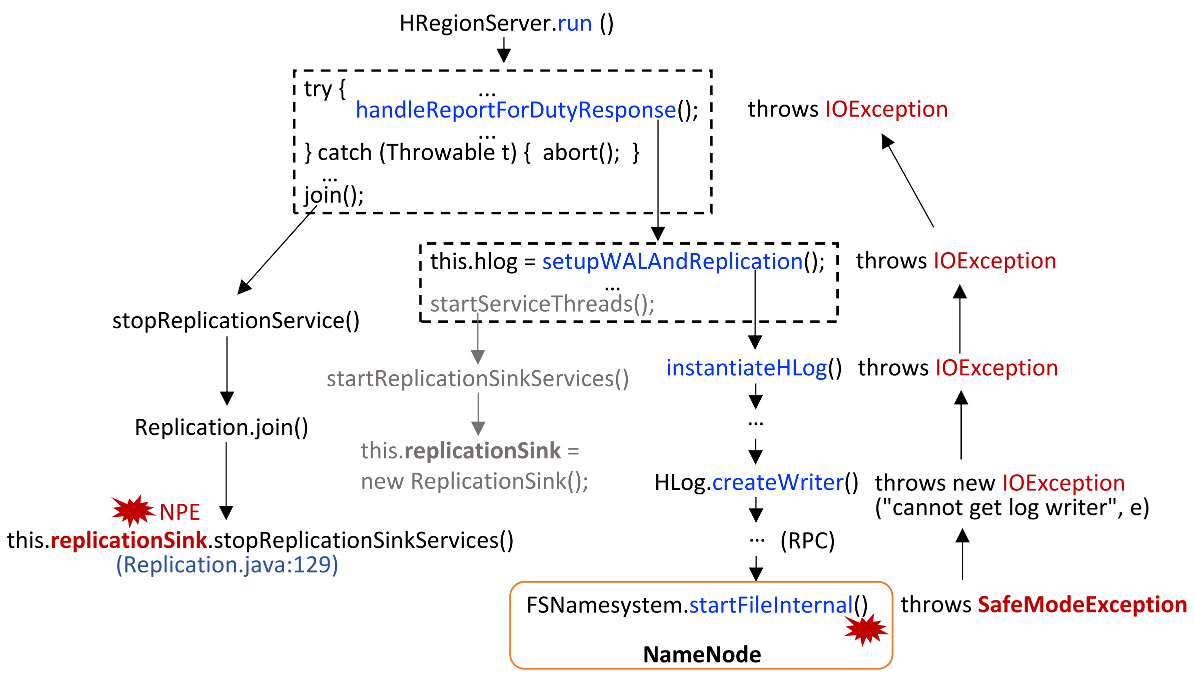

HRegionServer.run ()
try { ... handleReportForDutyResponse(); ... } catch (Throwable t) { abort(); } ... join();
throws IOException
this.hlog = setupWALAndReplication(); ... startServiceThreads();
throws IOException
stopReplicationService()
startReplicationSinkServices()
instantiateHLog()
throws IOException
Replication.join()
this.replicationSink = new ReplicationSink();
HLog.createWriter()
throws new IOException ("cannot get log writer", e)
NPE
this.replicationSink.stopReplicationSinkServices()
(Replication.java:129)
... (RPC)
FSNamesystem.startFileInternal()
NameNode
throws SafeModeException

Fig. 14. Bug-related code structure in hb8230.

Table 14
Different strategies and approaches for fixing failures.

Strategies & approaches for fixing failures		# Bugs
(a) retry	make retry possible (9)	14
	modify existing retry (5)	
(b) expose	throw early (5)	10
	fail fast (4)	
	only increase log level (1)	
(c) ignore	change original handling to ignore (6)	8
	add cleanup for existing ignore (2)	
(d) other handling corrections		3

timing problems, and about 70% of error-fixing patches tackle semantics problems.[18]

More interestingly, our study finds that ***one third*** (35 out of 106) of the distributed bugs are patched by eliminating end symptoms (i.e., failures) but **not root causes**. That is, the patches do not eliminate many intermediate errors (e.g., exceptions) that manifest before the end failures, and simply handle them in a better way to avoid the end failures. We will study these patches in more details below, and answer three research questions: (1) How do these symptom-fixing patches work? (Section 5.1.1) (2) How are network-connection IOExceptions (the most common type of exceptions in our studied distributed bugs) handled by these patches? (Section 5.1.2) (3) Are bugs fixed by symptom-fixing patches less or more important than those fixed by root-cause-fixing patches? (Section 5.1.3)

5.1.1. *How do symptom-fixing patches work?*

As shown in Table 14, these patches work in mainly three ways.

Retry For 14 bugs, the patches simply use retries to avoid the original failure, because the original cause of the failure is transient like unstable network connection, intermittent service issues from another node, etc. For nine of them, the retry mechanism is introduced by either surrounding the original buggy operations with a while loop, or changing the control flow so that the error handling code is redirected to use the existing retry mechanism. For five of them, the patches change the existing retry mechanisms, like the limit of retry numbers, the retry-partner node, etc.

Expose 10 bugs are fixed by better exposing intermediate errors to avoid the original severe failure symptoms. In five cases, the patches add an extra condition checking to allow throwing an exception earlier than before. In four cases, the patches simply change an existing exception handler's behavior into `abort` — by crashing the problematic node earlier, thus more severe failures like data losses are avoided. In one case (ca5725), the patch simply changes the logging level associated with an exception from DEBUG to WARN. Since this bug only occurs under very rare message timing, the developers feel the log-level upgrade is “by far the simplest fix and probably good enough in practice”.

Ignore For 8 out of 35 bugs, the patches are about how to better ignore the original failure-inducing exceptions. Specifically, for many of them, the patches simply remove the over-reaction (e.g., invocation of `abort`) in the original exception handler. For two bugs, the patches add some cleanup code like closing an invalid network socket before moving on to execute code after the exception handler.

5.1.2. *How are connection-related IOExceptions fixed?*

IOException[19] is the most common exception type in our study, encountered by ***more than one third*** (36 out of 106) of the distributed bugs. Particularly, about 60% of these IOExceptions (22 bugs) relate to network connection. Interestingly, among these connection-related IOExceptions, only five of them are eliminated by fixing the root causes, including timing causes and semantic causes. While *more than half* (in 12 bugs) of them are handled by retry strategy, three are handled by ignore strategy, and the remaining two are handled by the expose strategy (fail fast).

[18] These include very few memory problems like NULL dereferences.

[19] We only count standard java.io.IOException and its subclasses defined in Java library like java.net.NoRouteToHostException, not those defined by specific distributed systems like org.apache.hadoop.hbase.YouAreDeadException in HBase.

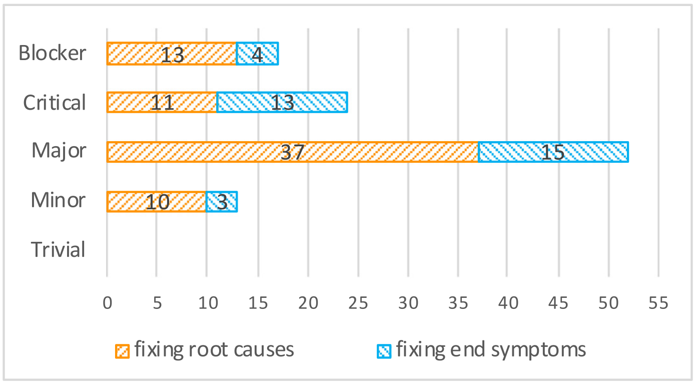


Fig. 15. Distribution of bug priority with different fixing types.

5.1.3. *Priorities of bugs fixed by different strategies*

In JIRA bug databases, every report is tagged with a priority level[20] (*blocker, critical, major, minor* or *trivial*) by developers, reflecting the relative importance of an issue. Fig. 15 shows the priority distribution among all bugs with different types of patch strategies. As we can see, whether developers decide to fix the root cause or only the log symptom of a bug does not depend on the priority level of the bug.

Finding 7. One third of the studied distributed bugs are patched by eliminating end symptoms (i.e., failures) but not root causes. Particularly, almost half of them are fixed by retry.

Implication. refers to the following summary.

Summary Interestingly, a significant portion of our studied distributed bugs are fixed by avoiding specific failure symptoms without eliminating root causes. Furthermore, these symptom-avoiding patches tend to follow a small number of standard strategies like retry, ignore, and exposing. Future research can explore the patterns or evidences for making such fixing decisions from other different aspects, and try to automatically generate these patches. And here are some insights we obtained from developers' perspectives for making such fixing decisions.

- ***Simple but work well enough in practice.*** Component failures like intermittent network failures and node crashes are common [24,25] in distributed systems, and are very difficult (sometimes impossible) to eliminate/fix, such as ca5725 we discussed in Section 5.1.1.
- ***Change as little as possible.*** Due to the large scale and complexity of distributed systems, developers are unwilling or feel unworthy to make distributed or protocol changes, even if the changes could eliminate some bug root causes. Therefore, the solution which minimizes the impact (the number of nodes to be changed and the extra burden brought by the change) of the patch is preferred. For instance, in hd1371, developers explicitly comment that they "do not want to have protocol change to just verify blocks for this extremely rare case of handicapped client" and picked "a light weighted solution with no need of protocol change."
- ***Bring issues into the spotlight when really necessary.*** Upon confirming that the exceptions do not affect the correctness of the system state, developers may "ignore" them rather than prevent them from being triggered, by logging these exceptions into lower-level log entries. Take mr3163 as an example, some INFOs with exceptions get printed in JobClient when killing a job, showing that JobClient fails to contact the corresponding AM of this job. "Looks like it's not just job kill, but anything that makes AM exit ungracefully, cause this error." Since the `kill` indeed takes effect, developers choose to"throw out the exceptions in DEBUG mode or when really necessary".
- ***When errors need more attention.*** This refers to the discussion on "*Expose*" in Section 5.1.1.

[20] Cassandra has migrated issues to new priority levels (urgent, high, normal, and low). To keep consistent with the other four projects, we still use previous priority for bugs in Cassandra.

5.2. *Are logs changed in bug patches?*

Since logs are concerned in our study, we wonder whether and how logs are changed when fixing distributed bugs. For *more than 20%* of (22 out of 106) the studied bugs, their patches make changes to log entries.

Logs in 16 (15%) bugs are enhanced for better debugging in the future. In 10 of them, either a new log entry (usually in DEBUG level) is supplemented, or a more detailed exception type will be printed. For example, in hd2905, an IOException will replace the previous NullPointerException to make the error more "friendly". In the other six bugs, more detailed information are added to existing log entries, such as different types of identifiers and the specific data size.

Six bugs' patches change the levels of some existing log entries. These are all symptom-fixing patches. Five of them lower log entry levels: with the patch, the same exception or intermediate error is not as severe as before. Meanwhile, based on the developers' discussion, we see that the lower-level log entries are still of significance. For example, in mr4448, the patch changes the YarnException into a WARN-level log message, instead of the original node abort indicated by a FATAL log entry. When being asked "if the log message is necessary", one of the developers explains "it's pretty important to log when the aggregation service isn't doing it's job properly, as it would help explain why logs are missing." One patch increases an existing log-entry level to help future debugging (i.e., the ca5725 bug discussed in Section 5.1.1).

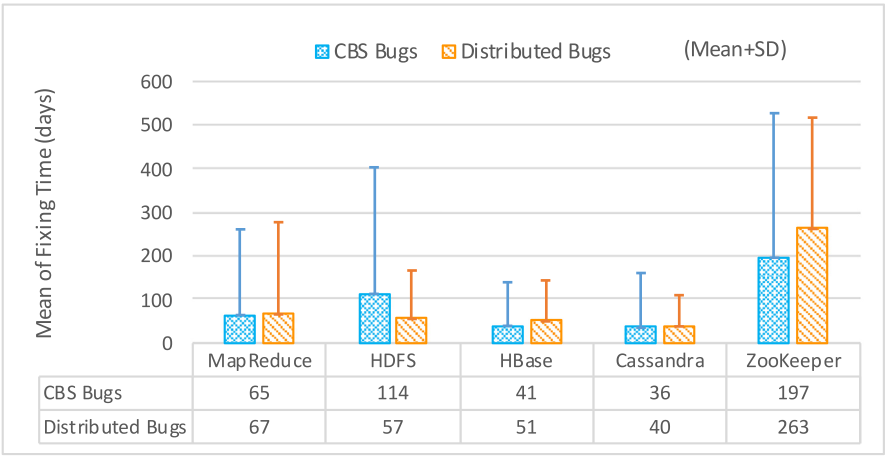
Summary Logs for a non-negligible portion (more than 20%) of studied distributed bugs are changed, either by enhancing log information to facilitate future debugging, or by changing log entry levels to help change failure symptoms. For the latter, the changes on logs follow routines and could be automated in the future.

5.3. *What is the relationship between logs and patches?*

Finally, we want to see how log symptoms (nodes) are connected with the patched nodes and methods for distributed bugs, and whether there are some opportunities for automated repair. Therefore, we try to answer two questions: (1) Are patched nodes the same as the symptom nodes (i.e., nodes that print out log symptoms discussed in Section 3)? (2) Are patched methods on the stack traces of log-symptom exceptions?

5.3.1. *Are patched nodes the same as symptom nodes?*

We studied all the 98 bugs whose failures are reflected by their logs to see if the patched nodes are the same as log-symptom nodes. Clearly, it is easier for developers to patch a bug once the failure is located if the answer is "yes".



	MapReduce	HDFS	HBase	Cassandra	ZooKeeper
CBS Bugs	65	114	41	36	197
Distributed Bugs	67	57	51	40	263

Fig. 16. Average fixing time (in days) of bugs in different applications.

Our study shows that, the patched nodes are exactly the same as symptom nodes for about 70% of the cases (68 out of 98); the patched nodes are completely different from the symptom nodes for about 20% of the cases (19 out of 98); the patched nodes are part of the symptom nodes in seven cases and are more than the symptom nodes in four cases.

Clearly, the latter three types, which contribute to about one third of the cases, pose challenges to patching. They are often the cases where the root cause or intermediate error in one node propagates to fail another node, or they are occasionally the cases where multiple nodes' cooperation is needed to fix a bug.

5.3.2. Are patched methods in logged stack traces?

76 distributed bugs' log symptoms contain exceptions.[21] We find that patches for about 40% of them (29 bugs) only fix methods on these exception stack traces. However, patches for about half of them (37 bugs) do not fix **any** method on these exception stack traces — the fixes are completely outside the exception-throwing call stack.

Summary Answers to this research question (RQ3.3) demonstrate the “scale” challenge of fixing distributed bugs. For about one third of the bugs, the patched nodes are not exactly the same (sometimes completely different) as the symptom nodes. Furthermore, the patches are more often outside the exception stack traces recorded in the logs. The first issue completely goes beyond the scope of bug-fixing research for traditional single-machine systems. The second issue calls for improvement over much previous work [26–29] that leverages exception stack traces to determine where to patch.

5.4. How quickly are distributed bugs fixed?

To answer how quickly distributed bugs are fixed, we collected the fixing time of bugs from when they are initially reported to when they are finally resolved. Further, we discuss whether there is correlation between the fixing time and some possible factors, such as bug priority and the fixing types (i.e., fixing root causes and fixing end symptoms).

Fig. 16 shows the average fixing time of the fixed CBS bugs and the studied distributed bugs, corresponding to the columns “# CBS Bugs” and “Distributed bugs” in Table 1, respectively. In general, the average fixing time of distributed bugs (68 days) is longer than that of the overall CBS bugs (64 days). Regarding different applications, fixing distributed bugs takes a longer time than fixing CBS bugs in all applications except HDFS. Although the fixing time also depends on the other factors such as bug priority, to some extent, the longer average fixing time can reflect the high complexity of the distributed bugs and the difficulty in debugging.

The priority of a bug reflects its importance compared with other bugs, which helps developers determine which bugs should be tackled first. Therefore, the bug priority is one of the important factors that affect the fixing time. Fig. 17 shows the average fixing time with different bug priorities, such as blocker, critical, major and minor. Trivial bugs are not included in the studied distributed bugs. The figure indicates that regarding different bug priorities, the fixing-time trend of distributed bugs is similar with that of CBS bugs, and the bugs in critical and major priorities take a longer time than the bugs in other priorities to be fixed. This also proves that bug priorities do not have necessary correlation with bug fixing time.

We also wonder whether the two fixing types discussed in Section 5.1 has correlation with the fixing time. Table 15 shows the average fixing time in different applications regarding different fixing types of the studied distributed bugs. In general, the average fixing time of the fixing-end-symptom bugs (75 days) is longer than that of the fixing-root-cause bugs (60 days). Fixing-end-symptom bugs need a longer time than fixing-root-cause bugs in all applications except MapReduce. This result indicates that fixing-root-cause bugs do not necessarily need longer fixing time than fixing-end-symptom bugs.

Summary On average, developers take a longer time to fix distributed bugs than the general CBS bugs. Although bug priorities and fixing strategies may affect the fixing time, our experimental results show that they do not have necessary correlation. Our study enlightens that the limitations of current system logs and debugging tools also have large effect on debugging efficiency and fixing time. For example, some key log entries are not provided when the bug is initially reported, or some timing-related information are not provided by the current logging mechanism. Sometimes, the proposed fixing method brings new problems and several rounds of discussion and testing are needed to generate the final patch. Therefore, future research can work on improving the quality of bug reports from the perspective of logs, and enhancing the timing-related output in the logging mechanism.

6. Discussion

In this section, we summarize the lessons learned and highlight the new research opportunities that are explored by our study.

Support for identifying global-symptom failures. As discussed in Findings 1 and 5, for a non-negligible part of distributed bugs, logs from more than one nodes are required to reflect the failure. However, it is quite challenging to identify such global symptoms, for example, cross-node repeating logs and cross-node inconsistency. For cross-node repeating logs, even in one node, the repeating pattern can be easily

[21] This is more than the “Exceptions in entries” column in Table 4, as exceptions also exist in other log-symptom patterns like repeated log entries, etc.

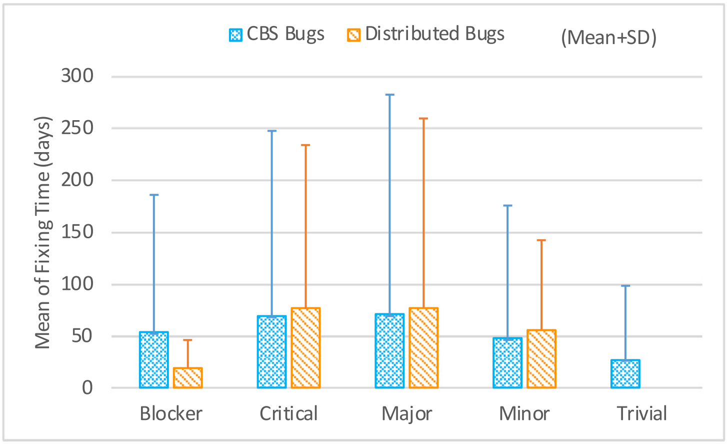


Fig. 17. Average fixing time (in days) of bugs in different priorities.

Table 15
Average fixing time (in days) of bugs with different fixing types.

App.\Fixing Type	Fixing Root Causes (# of bugs)	Fixing End Symptoms (# of bugs)
MapReduce	86 (19)	13 (7)
HDFS	19 (7)	89 (8)
HBase	12 (12)	63 (12)
Cassandra	35 (27)	62 (7)
ZooKeeper	215 (7)	598 (1)
Avg.	60 (71)	75 (35)

overwhelmed by the large amount of interleaved logs due to different threads and events. Therefore, the three repeating patterns summarized in Fig. 3 can help detect or filter the suspicious log entries. For both of the above two cross-node log symptoms, it is difficult to correlate the cross-node log entries for the same object in post-mortem log analysis, and developers need the help of log analysis tools to identify such implicit problems in distributed systems. Further, distributed tracing could also be employed to identify complicated global-symptom failures.

Note that Findings 3 and 4 shows the mismatch between the severity and the pre-defined log level of a log entry, which inspires us that when designing log analysis tools, low-level log entries also deserve attention, in conjunction with different contexts.

Log enhancement. The requirements for log improvement come from three aspects. (1) Identifying resource-related bugs. As discussed in Section 3.1, the bugs about resource (e.g., memory) leaks/exhaustions usually fail silently. No warnings or informative messages are printed when resource leak occurs until the program fails. (2) Diagnosing hang bugs and timing-related bugs. As discussed in Section 4.1, currently the log entries are not sufficient to make diagnosis for these two types of distributed bugs. For hang bugs, the thread or the method which leads to hang cannot be inferred from the log. For timing-related bugs, the current logs cannot provide the fine-grained timing, or it is difficult to figure out what exactly the objects or operations are involved in the race condition. (3) For better debugging in the future. As discussed in Section 5.2., logs in 15% of the studied bugs are enhanced for better debugging in the future, including different types of object attributes (such as identifiers and data size), and more detailed error information. Therefore, future research can work on the above directions to improve the log.

Automated patch generation. It is interesting that one third of the studied distributed bugs are patched by eliminating end symptoms rather than root causes (Finding 7). Moreover, these symptom-avoiding patches usually follow some standard strategies like retry, ignore and exposing. Future research is expected to explore the patterns to automatically generate these patches, along with the four insights we obtained from developers' perspectives for making such fixing decisions.

7. Related work

7.1. Bug studies in distributed systems

Several empirical studies have been conducted towards bugs in distributed systems recently [5,13,14,21,22,30], but none of them focus on debugging, fixing, or the role of software logs in debugging and fixing. Some of them focus on specific types of distributed system bugs, such as timing bugs [5], crash recovery bugs [21], configuration problems [30,31], network-partitioning faults [22], and timeout problems [17]. CBS [13] presented a broad study of all the issues that exist in a set of cloud systems during three years of period. For every issue, CBS checks what type of hardware problems (e.g., node crashes or limping hardware) or software problems (e.g., data race, configuration, or error handling) has caused the issue, which software component this issue belongs to, as well as the impact of the issue (e.g., security, reliability, or scalability). It does not study how a bug is diagnosed or fixed.

Yuan et al. [14] studied the bug-triggering and manifestation process in distributed systems, such as what type of input events and timing requirements tend to trigger bugs in distributed systems. Particularly, it finds that most catastrophic failures are caused by incorrect error handling, and hence builds a simple and effective rule-based static checker to detect incorrect error handlers, such as catch blocks that are empty, contain “to do”, etc. Our study gets a consistent finding with this previous study: exceptions are the most common failure log symptom. However, our study and the previous study have different focuses, and hence the characteristics we have studied are not covered by the previous study, like what log patterns (not only exceptions) and log levels tend to reveal not only failure occurrence but also failure types; how are distributed bugs, including those exception handling problems, are fixed, etc.

TaxDC [5] provided a taxonomy of distributed concurrency (DC) bugs, and mainly focuses on the triggering timing condition and input preconditions. Although the error and failure symptoms of DC bugs are studied, they are from different viewpoint (i.e. not correlated with the characteristics of logs), while we study the distributed bugs mainly from the perspective of logs. Regarding fixing, TaxDC only considers the fixing strategies of DC bugs, while our discussion covers more general distributed bugs, as well as the error handling strategies from different aspects.

In CREB [21], 103 crash recovery bugs were analyzed, which revealed some common vulnerabilities in crash recovery processes, such as the bug patterns of root causes, and triggering conditions. These crash recovery bugs belong to distributed bugs, but the focus in our study is different. In CREB, how to diagnose these crash recovery bugs is not concerned, and how these bugs are reflected through logs is not concerned, either. In contrast, our study concentrates on identifying the characteristics and providing debugging insights for general distributed bugs, not limited to the crash recovery bugs.

7.2. Log-based anomaly detection

Given the huge amount of system logs, several machine-learning based techniques have been proposed for log-based anomaly detection [32]. These techniques parse the unstructured logs, extract features, and feed them into various supervised (Decision Tree [33], SVM [34], and Logistic Regression [35]) or unsupervised (PCA [36], Invariant Mining [37], and Log Clustering [38]) learning models.

Based on our study (Section 3), the existing machine learning based techniques can effectively detect some anomalies that lead to repeated log entries, but would not perform well in detecting anomalies that lead to exceptions, which sometimes indicate failures and sometimes not, inconsistent log entries, or misleading log entries. We believe our study can help the future work along this direction.

7.3. Log-based diagnosis

Previous research has looked at how to diagnose functional [4,11] and performance bugs [8,9,12,39] using system logs. Our study has focused on functional bugs only, and future work will look at performance bugs too. SherLog [11] was designed to debug functional bugs in single thread execution, not for distributed bugs. Pensieve [4] helped reproduce failures based on logs. It is related to our root-cause diagnosis study (Section 4), but unrelated to other parts of our study. Pensieve and our study have consistent observations that IDs in logs are helpful and that debugging often needs to go beyond the symptom nodes. However, since reproducing failures is different from locating root causes, Pensieve does not use timestamps and log entry adjacency as we do in Section 4.

7.4. Log analysis platforms and tools

Among the existing automated log analysis platforms and tools, Splunk[22] and ELK stack [40] are two typical enterprise-level solutions. Splunk (known as the “Google for logs”) is one of the most popular commercial platforms for monitoring and analyzing systems logs. It treats logs as searchable text indexes and provides a powerful visual interface. But it only supports simple grep-like operations through a search language, and is not able to perform complex analysis [41,42]. ELK stack is a combination of three tools[23] — Elasticsearch (for deep search and data analytics), Logstash (for centralized logging management) and Kibana (for visualization), providing an open-source solution for log analysis. Some other popular open-source log analysis tools like Webalizer[24] and AWStats[25], due to the limitation of data scale, do not apply to process the large number of logs in cloud systems.

Although the above log analyzers may provide powerful query features on logs, they focus on log aggregation and filtering. Without knowing the characteristics of logs in reflecting failures, and the relationship between the logs and failures/errors, these tools are still not sufficient and cannot be utilized to discover and diagnose the complicated distributed bugs. Therefore, we believe that our work provides new perspectives, and can be complementary to the existing log analysis tools.

[22] https://www.splunk.com.
[23] https://www.elastic.co/products.
[24] http://www.webalizer.org.
[25] https://awstats.sourceforge.io.

8. Conclusion

Given the complexity and wide use of large-scale distributed systems, debugging distributed bugs has become crucial. Although much recent research has looked at how to detect and diagnose certain types of distributed bugs, particularly by leveraging run-time software logs, diagnosing and fixing general distributed bugs still remain an open problem, and there has been little empirical study that focuses on the various aspects of diagnosing and fixing general real-world distributed bugs. By carefully studying 106 real-word distributed bugs and their logs, seven new findings about combating distributed bugs are presented in our study. Our findings reveal the limitations of existing software logs and provide new research opportunities to develop techniques that can facilitate debugging in distributed systems. For example, some automated log analysis tools are expected to help developers identify the complicated log symptoms from a large number of system logs; the current logging mechanism still needs improvement, especially for resource, hang, and timing-related distributed bugs; it is useful and promising to generate the symptom-avoiding patches automatically. We have also contributed our in-depth analysis of the studied distributed bugs to the community. As part of our next-step work, we plan to develop some efficient log analysis tools to assist the debugging of distributed bugs, based on our findings.

Declaration of Competing Interest

No known competing financial interests or personal relationships.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful feedback and comments. This work was supported partly by National Key Research and Development Program of China under Grant No. 2016YFB1000804, partly by National Natural Science Foundation under Grant No. (61932007, 61972013, 61702024, 61421003).

References

[1] W.E. Wong, V. Debroy, A Survey of Software Fault Localization, Tech. Rep., Department of Computer Science, University of Texas at Dallas, 2009.
[2] Lloyd’s, 2018, (Cloud Down - The impacts on the US economy). https://www.lloyds.com/clouddown.
[3] P. Bailis, P. Alvaro, S. Gulwani, Research for practice: tracing and debugging distributed systems; programming by examples, Commun. ACM 60 (7) (2017) 46–49.
[4] Y. Zhang, S. Makarov, X. Ren, D. Lion, D. Yuan, Pensieve: non-intrusive failure reproduction for distributed systems using the event chaining approach, in: Proceedings of the Symposium on Operating Systems Principles (SOSP), ACM, 2017, pp. 19–33.
[5] T. Leesatapornwongsa, J.F. Lukman, S. Lu, H.S. Gunawi, TaxDC: a taxonomy of non-deterministic concurrency bugs in datacenter distributed systems, in: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, 2016, pp. 517–530.
[6] I. Beschastnikh, P. Wang, Y. Brun, M.D. Ernst, Debugging distributed systems: challenges and options for validation and debugging, Commun. ACM 59 (8) (2016) 32–37.
[7] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M.F. Kaashoek, Z. Zhang, D3S: debugging deployed distributed systems, in: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI), vol. 8, 2008, pp. 423–437.
[8] X. Zhao, Y. Zhang, D. Lion, M.F. Ullah, Y. Luo, D. Yuan, M. Stumm, lprof: a non-intrusive request flow profiler for distributed systems, in: Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), vol. 14, 2014, pp. 629–644.
[9] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, M. Stumm, Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle, in: Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), 2016, pp. 603–618.
[10] Q. Fu, J.-G. Lou, Y. Wang, J. Li, Execution anomaly detection in distributed systems through unstructured log analysis, in: Proceedings of the International Conference on Data Mining (ICDM), IEEE, 2009, pp. 149–158.

[11] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, S. Pasupathy, SherLog: error diagnosis by connecting clues from run-time logs, in: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), vol. 38, ACM, 2010, pp. 143–154.
[12] K. Nagaraj, C. Killian, J. Neville, Structured comparative analysis of systems logs to diagnose performance problems, in: Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI), USENIX, 2012. 26–26.
[13] H.S. Gunawi, M. Hao, T. Leesataponwongsa, T. Patana-anake, T. Do, J. Adityatama, K.J. Eliazar, A. Laksono, J.F. Lukman, V. Martin, et al., What bugs live in the cloud? A study of 3000+ issues in cloud systems, in: Proceedings of the Symposium on Cloud Computing (SoCC), ACM, 2014, pp. 1–14.
[14] D. Yuan, Y. Luo, X. Zhuang, G.R. Rodrigues, X. Zhao, Y. Zhang, P. Jain, M. Stumm, Simple testing can prevent most critical failures: an analysis of production failures in distributed data-intensive systems, in: Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), 2014, pp. 249–265.
[15] S. Lu, S. Park, E. Seo, Y. Zhou, Learning from mistakes: a comprehensive study on real world concurrency bug characteristics, in: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, 2008, pp. 329–339.
[16] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, C. Zhai, Bug characteristics in open source software, Empir. Softw. Eng. 19 (6) (2014) 1665–1705.
[17] T. Dai, J. He, X. Gu, S. Lu, Understanding real-world timeout problems in cloud server systems, in: Proceedings of IEEE International Conference on Cloud Engineering (IC2E), IEEE, 2018, pp. 1–11.
[18] J.-C. Laprie, Dependable computing: concepts, limits, challenges, in: Special Issue of the 25th International Symposium On Fault-Tolerant Computing, 1995, pp. 42–54.
[19] R.O. Suminto, A. Laksono, A.D. Satria, T. Do, H.S. Gunawi, Towards pre-deployment detection of performance failures in cloud distributed systems, Workshop on Hot Topics in Cloud Computing (HotCloud), USENIX, 2015.
[20] C. Zhang, J. Li, D. Li, X. Lu, Understanding and statically detecting synchronization performance bugs in distributed cloud systems, IEEE Access (2019).
[21] Y. Gao, W. Dou, F. Qin, C. Gao, D. Wang, J. Wei, R. Huang, L. Zhou, Y. Wu, An empirical study on crash recovery bugs in large-scale distributed systems, in: Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, 2018, pp. 539–550.
[22] A. Alquraan, H. Takruri, M. Alfatafta, S. Al-Kiswany, An analysis of network-partitioning failures in cloud systems, in: Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), 2018.
[23] R. Chmiel, M.C. Loui, Debugging: from novice to expert, ACM SIGCSE Bull. 36 (1) (2004) 17–21.
[24] J. Dean, Designs, lessons and advice from building large distributed systems, Keynote from LADIS, vol. 1, 2009.
[25] M.R. Mesbahi, A.M. Rahmani, M. Hosseinzadeh, Cloud dependability analysis: characterizing google cluster infrastructure reliability, in: International Conference on Web Research (ICWR), IEEE, 2017, pp. 56–61.
[26] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, M.J. Harrold, Fault localization and repair for java runtime exceptions, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), ACM, 2009, pp. 153–164.
[27] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, H. Mei, Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis, in: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2014, pp. 181–190.
[28] R. Wu, H. Zhang, S.-C. Cheung, S. Kim, CrashLocator: locating crashing faults based on crash stacks, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), ACM, 2014, pp. 204–214.
[29] L. Moreno, J.J. Treadway, A. Marcus, W. Shen, On the use of stack traces to improve text retrieval-based bug localization, in: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2014, pp. 151–160.
[30] S. Wang, C. Li, W. Sentosa, H. Hoffmann, S. Lu, Understanding and auto-adjusting performance-sensitive configurations, in: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, 2018.
[31] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, S. Pasupathy, Early detection of configuration errors to reduce failure damage, in: Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), 2016, pp. 619–634.
[32] S. He, J. Zhu, P. He, M.R. Lyu, Experience report: System log analysis for anomaly detection, in: Proceedings of the International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2016, pp. 207–218.
[33] M. Chen, A.X. Zheng, J. Lloyd, M.I. Jordan, E. Brewer, Failure diagnosis using decision trees, in: Proceedings of the International Conference on Autonomic Computing (ICAC), IEEE, 2004, pp. 36–43.
[34] Y. Liang, Y. Zhang, H. Xiong, R. Sahoo, Failure prediction in IBM BlueGene/L event logs, in: Proceedings of the International Conference on Data Mining (ICDM), IEEE, 2007, pp. 583–588.
[35] P. Bodik, M. Goldszmidt, A. Fox, D.B. Woodard, H. Andersen, Fingerprinting the datacenter: automated classification of performance crises, in: Proceedings of the European Conference on Computer systems, ACM, 2010, pp. 111–124.
[36] W. Xu, L. Huang, A. Fox, D. Patterson, M.I. Jordan, Detecting large-scale system problems by mining console logs, in: Proceedings of the Symposium on Operating Systems Principles (SOSP), ACM, 2009, pp. 117–132.
[37] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, J. Li, Mining invariants from console logs for system problem detection, in: USENIX Annual Technical Conference (ATC), 2010.
[38] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, X. Chen, Log clustering based problem identification for online service systems, in: Proceedings of the International Conference on Software Engineering (ICSE) Companion, ACM, 2016, pp. 102–111.
[39] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, T. Xie, Log2: a cost-aware logging mechanism for performance diagnosis, in: USENIX Annual Technical Conference (ATC), 2015, pp. 139–150.
[40] S. Chhajed, Learning ELK Stack, Packt Publishing Ltd, 2015.
[41] J. Stearley, S. Corwell, K. Lord, Bridging the gaps: joining information sources with Splunk., Workshop on Managing Systems via Log Analysis and Machine Learning Techniques (SLAML), USENIX, 2010.
[42] W. Shang, Bridging the divide between software developers and operators using logs, in: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, 2012, pp. 1583–1586.