Finding Heterogeneous-Unsafe Configuration Parameters in Cloud Systems

Sixiang Ma
The Ohio State University

Fang Zhou
The Ohio State University

Michael D. Bond
The Ohio State University

Yang Wang
The Ohio State University

Abstract

With the increasing prevalence of heterogeneous hardware and the increasing need for online reconfiguration, there is increasing demand for heterogeneous configurations. However, allowing different nodes to have different configurations may cause errors when these nodes communicate, even if the configuration of each node uses valid values.

To test which configuration parameters are unsafe when configured in a heterogeneous manner, this work reuses existing unit tests but runs them with heterogeneous configurations. To address the challenge that unit tests often share the configuration across different nodes, we incorporate several heuristics to accurately map configuration objects to nodes. To address the challenge that there are too many tests to run, we (1) "pre-run" unit tests to determine effective unit tests for each configuration parameter and (2) introduce pooled testing to test several parameters together. Our evaluation finds 41 heterogeneous-unsafe configuration parameters in Flink, HBase, HDFS, MapReduce, and YARN. We further propose suggestions and workarounds to make a subset of these parameters heterogeneous safe.

ACM Reference Format:
Sixiang Ma, Fang Zhou, Michael D. Bond, and Yang Wang. 2021. Finding Heterogeneous-Unsafe Configuration Parameters in Cloud Systems. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–28, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3447786.3456250

1 Introduction

While many distributed systems were initially designed under the assumption that all nodes share the same configuration, heterogeneous configuration has become increasingly popular for two reasons. First, heterogeneous hardware naturally calls for a heterogeneous configuration to achieve optimal performance [7, 18, 26, 28, 47]. Second, even for a homogeneous system, sometimes we need to change its configuration at run time to adapt to the workload, but rebooting the whole system with a new configuration may be too disruptive [5, 39]. To solve this problem, several approaches incrementally change the configuration of a subset of nodes, either by rebooting these nodes [8, 30, 35] or by utilizing application APIs [7, 14, 17, 27, 41], until all nodes have the new configuration. Both of these cases may cause different nodes to have different configurations, either in the long term or in the short term.

Heterogeneous configuration, however, may cause the system to fail if not used properly. For example, if one node is configured to encrypt its communication channel while the other node does not decrypt the messages, then unsurprisingly the communication will fail. This type of errors is different from the configuration errors caused by invalid configuration values [3, 36, 42, 43, 46, 48]: in our case, both configuration values (i.e., using and not using encryption) are valid; the problem is caused by two nodes with different configurations communicating with each other.

The goal of this paper is to investigate, in real-world applications, which configuration parameters cannot be set in a heterogeneous manner. We call them *heterogeneous-unsafe configuration parameters* in this paper. To achieve our goal, we have developed an approach to identify such parameters.

At a high level, our approach is not much different from classic program testing: we test the target application with different heterogeneous configurations and different inputs to see whether the application will fail. However, this method also encounters the classic challenge of program testing: a particular configuration parameter may only take effect when a particular piece of code is executed; thus, to test whether the parameter is heterogeneous-unsafe, we need to drive the application to a potential corner case.

To address this challenge, we observe that mature applications usually have well-designed unit tests, which have already considered this problem: to test the effects of a certain configuration parameter, some of these unit tests generate inputs so that the particular parameter will take effect and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSys '21, April 26–28, 2021, Online, United Kingdom
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8334-9/21/04...$15.00
https://doi.org/10.1145/3447786.3456250

have rules to check whether the application is in a healthy state. Following this observation, we utilize existing unit tests to find heterogeneous-unsafe parameters by assigning different configurations to different nodes in these tests.

We encounter two challenges when applying this idea. First, to run a unit test with a heterogeneous configuration, we need to be able to assign different configuration values to different nodes. This is trivial in a real distributed setting since we can give different configuration files to different nodes. This task, however, is significantly more challenging in unit tests, which often create nodes as threads within a process: on the one hand, a unit test may create a configuration object and share it with different nodes; on the other hand, a node may have subcomponents, which may create their own configuration objects. Both properties make it harder to map a configuration object to a particular node. To address this problem with minimal modification to the target application, we incorporate several heuristics to identify configuration sharing—in which case we clone the configuration object—and infer the mapping from configuration objects to nodes.

The second challenge is the large number of tests to run. To alleviate this problem, we incorporate several techniques: 1) we "pre-run" unit tests to identify which configuration parameters are used by each node type in each unit test, to avoid assigning a parameter to a node that will not use the parameter; 2) under the assumption that most parameters are safe, we introduce *pooled testing*, which tests several parameters together with one unit test, and separates them only if the pooled test fails.

Following these ideas, we have built *ZebraConf*, a framework to reuse existing unit tests to find heterogeneous-unsafe configuration parameters. We have applied ZebraConf to Flink [1], HBase [2], HDFS [16], MapReduce [32], and YARN [44]. Our evaluation yields the following results:

- ZebraConf reports a total of 57 heterogeneous-unsafe configuration parameters in these applications. Our manual analysis shows 41 of them are truly unsafe parameters, and the remaining 16 are false positives. While many of these unsafe parameters are expected (e.g., parameters related to encryption, compression, and heartbeat), some of them are more subtle. For example, we find setting a heterogeneous bandwidth limitation on different DataNode instances in HDFS can cause one DataNode with a high limit to overload a DataNode with a low limit, so that the latter cannot send progress reports in time, causing timeout. We further propose suggestions and workarounds to make a subset of these parameters heterogeneous safe.

- With its heuristics, ZebraConf correctly maps configuration objects to different nodes in 89.3% to 98.4% of the unit tests for each application. Achieving this level of correctness required adding or changing 21 to 38 lines of code to apply ZebraConf to each application.

- Pre-running tests and pooled testing reduce the total number of units tests to run by two to four orders of magnitude for each application. As a result, all tests can finish within 4,652 machine hours. While this number is certainly not small, it is affordable since we can run these tests in parallel (we used up to 100 machines in our experiments) and they do not need to be run frequently.

The rest of the paper proceeds as follows. Section 2 presents related work, which motivates our work. Section 3 gives an overview of our work. Sections 4–6 each introduce the design of one key component of ZebraConf. Section 7 presents the results of our evaluation and Section 8 concludes the paper.

2 Related Work and Motivation

Heterogeneous configuration. While many distributed systems were initially designed under the assumption that all nodes have the same configuration (i.e., homogeneous configuration), heterogeneous configuration has become increasingly popular for several reasons.

First, heterogeneous hardware naturally calls for a heterogeneous configuration to achieve the best performance [7, 26, 28, 47]. For example, many systems allow the administrator to configure the number of threads, the size of memory, or the bandwidth limitation of each node, and such configurations naturally depend on the hardware setting of each node.

Second, even for a homogeneous system, it is often beneficial to reconfigure the system to adapt to the workload [7, 27, 30, 41]. While rebooting the whole system with a new configuration is always possible [4, 10, 22, 29, 50], it is often too disruptive especially for a large cluster. To solve this problem, recent works propose to incrementally reconfigure a subset of nodes, either by rebooting these nodes (i.e., rolling restart) [5, 8, 30, 33, 35] or by utilizing application APIs [12, 14, 17, 25, 38], until all nodes are reconfigured.

For both cases, since it is often hard to determine the optimal configuration values when booting the system, a number of systems (e.g., Kafka [24], HBase [2], HDFS [16], and Redis [37]) provide APIs to allow the administrator to change certain configuration parameters of a node at run time. For example, HDFS parameter `dfs.datanode.balance.bandwidthPerSec` was made online reconfigurable starting from HDFS 0.20 [18]. The introduction of these APIs indicates a strong motivation to reconfigure nodes at run time, presumably leading to more heterogeneous configurations in the future.

While heterogeneous configuration is beneficial and arguably unavoidable, it may lead to correctness issues when nodes with different configuration values communicate. Some of these issues are obvious: if a node is configured to encrypt its data and another node is not aware that data is encrypted, they cannot communicate properly. Some of these issues are more subtle and perhaps unexpected as shown in our evaluation. A goal of this work is to find such heterogeneous-unsafe configuration parameters in real-world applications.

Finding configuration errors. A substantial amount of work targets identifying configuration errors caused by invalid configuration values (including but not limited to [3, 23, 36, 42, 43, 46, 48]). For example, ConfValley [23] defines a systematic way to validate configuration values. PCheck [43] extracts application code that checks the validity of configuration values and executes such code before deployment.

Our work is different from these prior works because a parameter can be heterogeneous unsafe even if every value at different nodes is valid (e.g., one node is configured to encrypt data and another node is configured not to encrypt data). Therefore, it is impossible to check such problems locally at one node.

3 Design Overview

3.1 Goal and Definitions

This work targets a distributed system, which is composed of multiple nodes (i.e. processes). We assume each node can be configured independently with its own configuration file.

To formally define heterogeneous-unsafe configuration parameters, we first introduce the following definitions:

- F denotes a configuration file and $F(p)$ denotes the value of parameter p in the configuration file.
- $HomoConf(F)$ denotes a homogeneous configuration, in which all nodes have the same configuration file F.
- $HeteroConf(F_1, ..., F_n)$ denotes a heterogeneous configuration, in which node i has configuration file F_i.
- I denotes a sequence of inputs to the target application. Inputs include explicit inputs through the application APIs, as well as all nondeterministic factors, such as timing and randomness, which are modeled as implicit inputs.
- A testing *oracle* can verify whether the application is in a correct state given I and either a $HomoConf(F)$ or a $HeteroConf(F_1, ..., F_n)$.

We assume that within any given node, all code always sees the same configuration. In other words, issues caused by incorrect implementation of online reconfiguration (e.g., missing updates to some variables depending on the parameter to be reconfigured) are outside the scope of our work.

Definition 3.1 (Invalid heterogeneous configuration)**.** *We say $HeteroConf(F_1, ..., F_n)$ is invalid if $\exists I$ such that*

$$\neg\, oracle(I, HeteroConf(F_1, ..., F_n)) \;\wedge\; \forall_{i \in \{1,...,n\}}\, oracle(I, HomoConf(F_i))$$

Intuitively, this means that an invalid heterogeneous configuration is one that causes problems even if every configuration file is individually valid.

Definition 3.2 (Heterogeneous-unsafe configuration parameters)**.** *We say a set of parameters P is heterogeneous unsafe if*

- *there exists an invalid heterogeneous configuration $HeteroConf(F_1, ..., F_n)$ in which for every parameter in P, at least two configuration files have different values of the parameter, and all other parameters have the same value in all configuration files, i.e.,*

$$\forall_p \left(p \in P \Longleftrightarrow \exists_{i,j}\, F_i(p) \neq F_j(p) \right)$$

and

- *P is minimal, i.e., there does* not *exist $P' \subsetneq P$ that satisfies the above condition.*

Intuitively, this defines the minimal set of parameters that may cause invalid heterogeneous configurations when given different values. If parameters do not depend on each other, this definition can further be simplified to be on individual parameters. The goal of this work is to identify heterogeneous-unsafe configuration parameters in real-world applications.

3.2 Our Approach

To understand whether certain configuration parameters are heterogeneous unsafe, we use the traditional software testing approach: we generate a number of heterogeneous configurations, each with different values of the target parameters; we then run the target application with these heterogeneous configurations and different inputs, and check whether the application encounters errors. However, the challenge of this approach is that a particular configuration parameter may only take effect when rarely executed code is executed, and thus when testing the parameter, we need to generate specific inputs to drive the application to the corner case.

To address this challenge, we utilize existing unit tests built by the application developers: we run these unit tests with the corresponding heterogeneous and homogeneous configurations. Since the unit tests of a mature application should cover most of an application's code [40, 49], they naturally provide the ability to drive the application to corner cases and test whether the application is in a correct state. In other words, we assume the unit tests can provide the input I and approximate the *oracle* in Definition 3.1. Following this idea, we have built ZebraConf, a framework to generate heterogeneous configurations, to run unit tests with these configurations, and to modify the target application to facilitate such testing.

Unit tests and integration tests. Traditionally, "unit tests" refer to tests that target individual functions or components of a system, and thus cannot be used for our purpose since they do not start multiple nodes. In contrast, "integration tests" refer to tests that target the whole system. However, to simplify testing, today's open-source software often implements its whole-system tests by running nodes as threads in one process and managing these tests as unit tests (e.g., MiniDFSCluster in HDFS, MiniCluster in Flink, etc). ZebraConf targets reusing such whole-system unit tests. ZebraConf should be able to reuse integration tests as well, since reusing integration tests is simpler than reusing unit tests (Section 6.1), though we have not done any experiments with integration tests.

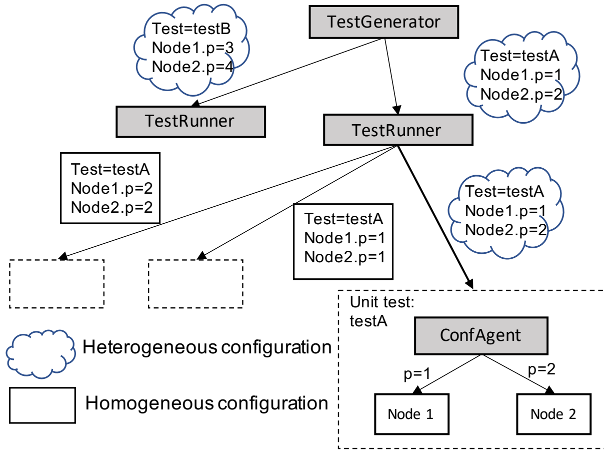

Figure 1. Overview of ZebraConf.

3.3 System Architecture

As illustrated in Figure 1, ZebraConf consists of three key components: TestGenerator, TestRunner, and ConfAgent.

At the top layer, TestGenerator determines which unit tests to run and what heterogeneous configurations to use for each unit test.

At the middle layer, given a unit test and a heterogeneous configuration, TestRunner follows Definition 3.1 to test 1) whether the unit test reports an error for the given heterogeneous configuration; and 2) whether the unit test reports an error for any corresponding homogeneous configuration.

At the bottom layer, ConfAgent is responsible for running a given unit test with a given configuration, either heterogeneous or homogeneous. The main task of ConfAgent is to distribute different configurations to different nodes.

4 Design of TestGenerator

TestGenerator is responsible for generating all the test instances. In the general case, a test instance is represented by a tuple of a unit test and a heterogeneous configuration $HeteroConf(F_1, F_2, ..., F_n)$. Table 1 shows the number of unit tests and parameters in different applications. As one can imagine, enumerating the combination of all parameter values and all unit tests will generate too many test instances Exacerbating this problem, we observe that many whole-system unit tests can take a long time (e.g., several minutes), because they need to wait for a cluster to be set up. To alleviate this problem, we introduce a number of strategies and techniques:

Test parameters independently. We assume that whether a configuration parameter is unsafe when set in a heterogeneous manner (i.e., different values on different nodes) does not depend on the values of other parameters. This assumption allows us to simplify Definition 3.2 to test each parameter individually rather than testing their combinations, which greatly reduces the number of test instances. With this strategy, TestGenerator converts the representation of a test instance into

	#Unit tests	# App-specific parameters
Flink	26,226	447 [11]
Hadoop Tools	1,518	N/A
HBase	4,985	206 [15]
HDFS	6,445	579 [20, 21]
MapReduce	1,423	210 [31]
YARN	4,806	465 [45]

Table 1. Statistics for each application. Hadoop Tools provide a number of tools to support other applications, but do not have their own parameters. All other applications have their own parameters (see table's citations for details) and share the Hadoop Common library (see [13] for details), which has 336 parameters.

a tuple of 1) a unit test, 2) the name of the parameter to test, and 3) the parameter value at each node. All other parameters will use the original values in the particular unit test.

Of course this assumption does not always hold, and TestGenerator allows additional rules to specify that when testing parameter p_1 with value v_1, we should set p_2's value to v_2. Currently TestGenerator requires the developer's effort to generate these rules and in our experiments, we manually add rules for a few parameters which obviously depend on others. For example, in HDFS there is a parameter to configure whether to use the http or https protocol, and two parameters to set the http and https addresses. Following the HDFS documentation, we set the http address if using the http protocol and set the https address if using the https protocol. Future work could extract the relationship between different parameters automatically, by relying on parameter dependence analysis [6].

Select parameter values to test. For boolean parameters, selecting values is trivial since we only need to test true and false values. For other types of parameters, we manually select a few values that we believe are representative based on the documentation of the target application. For numerical values, apart from the default value, we select one that is much larger than the default value, one that is much smaller, and values that have specific meanings (e.g., 0 or -1 sometimes means this feature is disabled). For string values, we select the values listed in the documentation of the target application.

Select representative value assignment. If a unit test contains n nodes and we need to test a parameter with two different values v_1 and v_2, then there are 2^n ways to assign values to nodes. To reduce this number, we select a few representative assignment strategies, based on the observation that nodes of the same type are executing the same piece of code and thus are mostly symmetric. We first divide nodes into groups based on their types and then test each group G with the following strategies: 1) assign v_1 (v_2) to all nodes in G and assign v_2 (v_1) to all other nodes. This strategy tests heterogeneous configuration across different types of nodes; 2) assigns values in

a round robin order to nodes within G (i.e., assign v_1 (v_2) to the first node, assign v_2 (v_1) to the second node, assign v_1 (v_2) to the third node, and so on), and assign v_2 (v_1) to all other nodes. This strategy further tests heterogeneous configuration within nodes of the same type.

Pre-run unit tests. The previous step generates a list of test instances, each specifying how to assign configuration values to different nodes in a unit test. However, not all these test instances are effective to test a heterogeneous configuration, and TestGenerator "pre-runs" all unit tests once to filter ineffective test instances.

First, many unit tests do not create any nodes: as explained in Section 3.2, the term "unit test" initially referred to tests targeting individual functions and has changed recently to include whole-system tests. Unit tests that do not create any node are of course unable to test heterogeneous configurations. During the pre-run of a unit test, if the test does not start any node, then TestGenerator removes the test from its list.

Second, not all nodes in all unit tests use all parameters. If we assign a parameter value to a node not using the parameter, then of course we are wasting time. This fact provides an opportunity for us to further trim the number of tests to run. To exploit this opportunity, during the pre-run TestGenerator records which node is using which parameter in each unit test. When generating test instances, TestGenerator applies the following rule: for a unit test with nodes of type A and a parameter p, TestGenerator will only generate test instances to test p on nodes of type A if these nodes actually use p in the pre-run. For example, in HDFS, TestGenerator will not test `dfs.datanode.balance.bandwidthPerSec` on NameNode because NameNode never uses this parameter.

A challenge of this technique is how to determine whether a node "uses" a parameter. In our current implementation, we define "use" as reading a parameter, which is easy to implement but is conservative since a node may read a parameter's value during initialization and never use the value later. Future work could explore using program analysis to improve the accuracy of identifying whether a parameter's value is used.

As shown in our evaluation (Table 5), pre-running unit tests and filtering ineffective tuples allows us to reduce the number of unit tests to run by up to three orders of magnitude.

Pooled testing. To further reduce testing time, we observe that most configuration parameters are heterogeneous safe. This motivates us to use a divide-and-conquer approach: instead of testing only one parameter for heterogeneous safety when running a unit test, we test multiple parameters (called a "pool") together. If the unit test does not report any errors, then all of these parameters are assumed to be safe; otherwise, we divide these parameters into two groups and test each group recursively, until we can identify all unsafe parameters. In our evaluation, we set the maximal pool size to be equal to the number of parameters.

In order for this approach to be an effective optimization, we need most pools of parameters to be error free. However, we found that the efficiency of this approach is hampered by a small number of heterogeneous-unsafe parameters that fail almost every unit test. Examples include parameters related to encryption and compression, which are used by most unit tests. To solve this problem, if TestGenerator finds that a parameter has failed many unit tests, TestGenerator will mark the parameter as unsafe and avoid using it in future tests.

Test in parallel. Unit tests are independent from each other, which provides a natural opportunity to run unit tests in parallel. In our experiments, we run unit tests on a cluster of machines to reduce total wall-clock time.

5 Design of TestRunner

TestRunner is responsible for running a test instance (a unit test and a heterogeneous configuration) generated by TestGenerator. Based on Definition 3.1, TestRunner will test both the heterogeneous configuration generated by TestGenerator and all corresponding homogeneous configurations: if the former one reports an error and the latter ones do not, then TestRunner will report a heterogeneous-unsafe parameter.

TestRunner's task is complicated by nondeterministic errors in unit tests. For example, if a heterogeneous configuration has a probability to fail but does not fail in one test, then we may miss a heterogeneous-unsafe configuration parameter (i.e., false negative). If one of the homogeneous configurations has a probability to fail but does not fail in one test, then we may report a heterogeneous-safe parameter as unsafe (i.e., false positive). In our experiments, we find that false positives caused by nondeterministic errors are common.

To reduce false positives in the face of nondeterminism, we run multiple trials of a test instance (both its heterogeneous configuration and corresponding homogeneous configurations) until we can be sure that the parameter is heterogeneous unsafe with high probability, according to hypothesis testing using a significance level of 0.0001 (i.e., $1 - 99.99\%$).

To minimize run time, we run multiple trials of a test instance only if its heterogeneous configuration fails and none of its homogeneous configurations fail in the first trial. This approach saves time but can result in false negatives due to nondeterminism. To reduce false negatives, a developer would need to run the test instances multiple times, which is not ideal but is the standard solution for most nondeterministic errors. On the other hand, although we run only one trial for most of the test instances, most parameters are tested by multiple test instances, reducing the chances of false negatives.

6 Design of ConfAgent

In ZebraConf, ConfAgent is responsible for running a unit test with a given configuration. Since the major challenge comes from heterogeneous configurations, our discussion focuses on this context, using an example shown in Figure 2.

```
1  /* Blank constructor */
2  public Configuration() {
3    ConfAgent.newConf(this);
4    ...
5  }
6
7  /* Clone constructor */
8  public Configuration(Configuration other) {
9    ConfAgent.cloneConf(other, this);
10   ...
11 }
12
13 /* Get the value of name parameter */
14 public String get(String name) {
15   String value = properties.getProperty(name);
16   return value;
17   return ConfAgent.interceptGet(this, name);
18 }
19
20 /* Set the value of the name parameter */
21 public void set(String name, String value) {
22   ConfAgent.interceptSet(this, name, value);
23   ...
24 }
```

(a) Pseudocode of the Configuration class.

```
1  private Configuration conf;
2  private Component c;
3
4  public static void main() {
5    ...
6    /* create a blank config object */
7    Configuration conf = new Configuration();
8    Server server = Server(conf);
9    ...
10 }
11
12 /* Server init function */
13 public Server(Configuation conf) {
14   ConfAgent.startInit(this, 'Server');
15   /* replace saving reference with refToClone
        */
16   this.conf = conf;
17   this.conf = ConfAgent.refToCloneConf(conf);
18   /* initialize this Server */
19   c = new Component();
20   ...
21   ConfAgent.stopInit();
22 }
23
24 protected void funA() { ...}
```

(b) Pseudocode of the Server class.

```
1  private Configuration conf;
2
3  /* Component init function */
4  public Component() {
5    this.conf = new Configuration();
6    ...
7  }
```

(c) Pseudocode of the Component class.

```
1  public void test () {
2    Configuration conf = new Configuration();
3    /* create servers */
4    Server server1 = new Server(conf);
5    Server server2 = new Server(conf);
6    ...
7    server1.funA();
8    System.out.println(conf.get(XXX));
9  }
```

(d) Pseudocode of a unit test.

Figure 2. An example of how an application and its unit tests may use configuration objects, and how to modify the application to support ZebraConf with the ConfAgent API (lines with `ConfAgent` are added by the developer).

6.1 Challenges

To run a unit test with a heterogeneous configuration, ConfAgent needs to be able to control the configuration values at each node. This would be trivial in a real distributed setting or in an integration test, in which each node would be running as a process: we could give each node a separate configuration file. However, the context of unit tests is significantly more challenging because unit tests often create nodes as threads within a single process, and all nodes inherently share the same configuration file, making it infeasible to assign different configuration files to different nodes.

To address this problem, we observe that well-designed applications usually keep track of configuration values in a dedicated configuration object (e.g., Figure 2a). The configuration object usually provides a *get* function to retrieve a certain configuration value and a *set* function to set the value. Therefore, if we can modify the configuration objects to return different values to different nodes, we will achieve our goal. This approach has the benefit that it only requires modifying a dedicated class. The challenge is how to determine which node is calling the *get* function of a particular configuration object. To illustrate the challenge, we describe a few approaches we tried that failed.

Determine caller based on configuration object. Initially, we thought that if each node uses one configuration object internally, then our task would be trivial: we could annotate the creation of each configuration object to connect it to a node. However, when investigating real applications and their unit tests, we found that this assumption was almost never true, manifesting in two different ways. First, a unit test often creates a configuration object by itself and then shares the object with different nodes. For example, as shown in Figure 2d, the unit test creates a `Configuration` object, and then uses the object to create two `Server` objects. In this case, these two `Server` objects and the unit test (the unit test itself is treated as a "client" node in ZebraConf) are sharing the

`Configuration` object. In our experiments, we find configuration object sharing occurs in 99.9%, 99.8%, 96.5%, 100%, and 88.5% of the unit tests that involve configuration usage in Flink, HBase, HDFS, MapReduce, and YARN respectively. Second, sometimes a node creates multiple configuration objects, which violates our assumption as well. For example, at line 19 in Figure 2b, a `Server` object creates a `Component` object, which later creates its own `Configuration` object (line 5 in Figure 2c).

Determine caller based on object allocation chain. In our second attempt, we tried to tie each Java object to a node. Then if a Java object called `Configuration.get`, we could know which node was making the call. To implement this idea, we first annotated a few objects as roots, which are typically the main object of a node (e.g., DataNode and NameNode in HDFS). Then we applied the following rule: if object A's method creates object B, then A and B belong to the same node. While we found no correctness problems in this approach, it was too invasive: we needed to add a `node` field to each object; we needed to modify the constructor of each object to pass the `node` field from its creator; and we needed to do this not only for the target application, but also for any third-party libraries used by the application. As a result, this approach incurred too much overhead, in terms of both the effort to modify the application and the CPU and memory usage at run time.

Determine caller based on calling thread. In the third attempt, we implemented a simplified version of our second attempt by only keeping track of which node each *thread* belongs to. We annotated the main thread of a node as the root and applied the following rule: if thread A creates thread B, then A and B belong to the same node. Then whenever `Configuration.get` was called, we could retrieve the thread ID of the caller and infer which node was making the call. Compared to the second attempt, this approach was simpler since getting the thread ID whenever `Configuration.get` was called did not require tracking the object allocation chain from the thread to `Configuration.get`. However, this approach relied on the assumption that a node's code is only executed by its own threads, and once again, we found that the design of unit tests violates this assumption: for testing convenience, it is common for a unit test to directly call nodes' internal functions for various purposes, such as stopping a node, adding data, checking status, and injecting faults. As a result, a node's code may be called by the unit test thread (i.e., main thread), and thus we cannot determine which node is calling `Configuration.get`. For example, in Figure 2d, line 7, the unit test calls an internal function `funA` of `server1`. In this case, `funA` should use the configuration object of `server1`, but determining the caller based on the calling thread would instead use the configuration object of the unit test.

Application	Types of nodes
Flink	JobManager, TaskManager
HBase	HMaster, HRegionServer, ThriftServer, RESTServer
HDFS	NameNode, DataNode, SecondaryNameNode, JournalNode, Balancer, Mover
MapReduce	MapTask, ReduceTask, JobHistoryServer
Yarn	ResourceManager, NodeManager, ApplicationHistoryServer

Table 2. The types of nodes we investigated.

6.2 ConfAgent's Solution

ConfAgent's solution is based on our first attempted solution (determining the caller based on the configuration object). To achieve high accuracy and to minimize the modification to the target application despite the two challenges (i.e., configuration object sharing across different nodes and multiple configuration objects within a node), ConfAgent incorporates several heuristics, based on our observation of how configuration objects are used in the unit tests and the applications.

Observation 1: the number of types of nodes is small. As discussed previously, for all methods, we need to define certain "root" classes or objects to separate different nodes at run time. Fortunately we find this is a simple task: all the applications we investigated have a well-defined node class for each type of node, e.g., NameNode and DataNode in HDFS. The number of types of nodes is small, which means manual annotation is feasible. Table 2 records the node types we picked in our work.

Observation 2: flow of configuration objects. We observe that information about configuration objects can flow in three ways, providing hints about how to track them.

- *Observation 2.1: creating a new blank configuration object.* Both the unit test itself and a node may create new configuration objects (e.g., line 2 in Figure 2d and line 5 in Figure 2c). We observe that a node usually creates its configuration objects in an initialization function, typically the constructor function or another `init` function in the node class. This means we can annotate the initialization function to map a configuration object to its node. Note that sometimes the object is created by a function called by the initialization function, instead of by the initialization function itself (e.g., line 19 in Figure 2b calls the constructor of the `Component` class, which creates a new `Configuration` object). And sometimes a node may create multiple configuration objects, usually for its subcomponents. To capture such relationships, ConfAgent adds the following rule: if a configuration object is created at time t on thread A, and thread A executes the initialization function of a node between t_1 and t_2, and $t_1 < t < t_2$, then the configuration object belongs to the particular node.

- *Observation 2.2: cloning a configuration object.* Both the unit test and the application may clone a configuration object by creating an object with a constructor that copies values from an existing object. We observe that the original object and the cloned object usually belong to the same entity (i.e., a node or the unit test itself).
- *Observation 2.3: creating a new reference to a configuration object.* This will not create a new object, and thus will not affect the mapping from configuration objects to nodes, except in the following case: we observe a node's initialization function often takes a configuration object as an argument and assigns it to an internal reference (line 16 in Figure 2b). In a real distributed setting, the main function of the node class will create the configuration object and use it to create the node class (lines 7–8 in Figure 2b), but in the unit test, the unit test itself replaces the main function and may share the configuration object with many nodes (lines 2–5 in Figure 2d). To solve this problem, we require the developer to replace such a configuration object reference in the initialization function with a clone (lines 16–17 in Figure 2b).

Observation 3: exceptions to the previous observations could lead to a high false positive rate. Exceptions to the previous observations may cause ConfAgent to fail to assign proper values to different configuration objects, leading to false positives. In particular, if ConfAgent assigns different values to configuration objects within the same node, it may cause errors that will not happen in a real distributed setting, since a node in a distributed setting will read the same value for the same parameter from a configuration file. Although the total number of exceptions is small compared to the total number of unit tests, the number of heterogeneous-unsafe configuration parameters is small as well, so these exceptions would lead to a high false positive rate if they were not filtered properly. ConfAgent tries to identify such cases during the pre-run: for each unit test, it tries to map each configuration object to either a node or the unit test itself; if it ultimately finds that a configuration object is mapped to no entity, then for this unit test, ZebraConf avoids testing any parameters used by the unidentifiable configuration object.

Based on such observations, ConfAgent works as follows: first, the developer needs to annotate the node initialization and configuration object creation with the API provided by ConfAgent (see Section 6.3). Then at run time, ConfAgent follows the following rules (based on the observations above) to determine the mapping from configuration objects to nodes:

Rule 1.1: Configuration object creation (Observation 2.1). If a configuration object is created at time t on thread A, and thread A executes the initialization function of a node between t_1 and t_2, and $t_1 < t < t_2$, then the configuration object "belongs to" the particular node.

Rule 1.2: Configuration object creation. If a configuration object is created when no node has initialized, then we say that this object "belongs to the unit test."

Rule 2: Configuration object reference (Observation 2.3). If the developer replaces a configuration object reference with a clone during initialization, then the object to be cloned belongs to the unit test, and the cloned object belongs to the node that executes the initialization function.

Rule 3: Configuration object clone (Observation 2.2). If a configuration object is cloned from another configuration object but not by Rule 2, then these two objects belong to the same entity.

As mentioned previously, if ConfAgent fails to map a configuration object in a unit test with these rules during the pre-run, ConfAgent excludes the test instances that combine the unit test and the parameters used by the unidentifiable configuration object from further testing, since they may generate false positives. Our evaluation shows that for four out of the five target applications, less than 5% of the test instances are excluded; for the remaining one, about 10% of the instances are excluded. Such numbers indicate that these rules accurately cover a high percentage of unit tests.

6.3 ConfAgent API and Implementation

To implement the aforementioned rules, ConfAgent provides an API for the developer to annotate the source code of the target application:

- `startInit(node, nodeType)` and `stopInit()`. The developer should use these two methods to annotate the start and the end of the initialization function (e.g., lines 14 and 21 in Figure 2b). They serve to implement Rule 1.1.
- `newConf(conf)`, `cloneConf(origConf, newConf)`, and `refToCloneConf(origConf)`. These three methods are for tracking configuration objects, which are used to implement all of the rules mentioned above. The developer can annotate the constructor of the configuration class with `newConf` or `cloneConf` (e.g., lines 3 and 9 in Figure 2a). The developer should use `refToCloneConf` to replace a reference to a configuration object with a clone in the node initialization function (e.g., lines 16–17 in Figure 2b).
- `interceptGet(conf, paraName)` and `interceptSet(conf, paraName, paraValue)`. These two methods are for intercepting the get and set functions of configuration objects, in order to implement the heterogeneous configuration. The developer can place these two methods in the get and set functions of the configuration class (e.g., lines 17 and 22 in Figure 2a).

To implement these API methods, ConfAgent maintains the following data structures:

- A `nodeTable` object records the following for each node: `nodeID` is the hashCode of the corresponding node; `nodeType` is the type of the node (e.g., NameNode or DataNode);

nodeIndex is *i* if the node is the *i*th node of nodeType, which is used by the TestGenerator to assign a configuration value to a particular node (note that TestGenerator cannot use nodeID for this purpose since nodeID may not be consistent across multiple runs); confIDs is an array recording the hashCode of all configuration objects belonging to this node; parentConfID records the hashCode of the configuration object passed as the argument to the initialization function, if any.

- A unitTestConfIDs list records configuration objects belonging to the unit test. An uncertainConfIDs list records configuration objects that cannot be mapped to anywhere.
- The parentTochild<childConfID, parentConfID> map keeps track of all cloning relationships.
- The threadContext<ThreadID, nodeID> map keeps track of whether an initialization function of a particular node is executing on a thread.

When startInit(node, nodeType) is called, ConfAgent puts a new entry in nodeTable (confIDs is empty and parentConfID is null). It further puts the current thread ID and node ID in threadContext. When stopInit is called, ConfAgent removes the current thread ID from threadContext.

When newConf, cloneConf, or refToCloneConf is called, ConfAgent updates the information based on the rules mentioned previously:

- When newConf(conf) is called, if no node has initialized yet, ConfAgent puts conf in unitTestConfIDs (Rule 1.2). If threadContext has a pair of <ThreadID, nodeID> for the current thread, ConfAgent puts conf into nodeTable (Rule 1.1); otherwise, ConfAgent puts conf in uncertainConfIDs.
- When cloneConf(origConf, newConf) is called, ConfAgent searches if either origConf or newConf already belongs to a node (i.e., in nodeTable) or the unit test (i.e., in unitTestConfIDs): if so, ConfAgent puts the other one in the same group (Rule 3); otherwise, ConfAgent puts both configuration objects in uncertainConfIDs. In either case, ConfAgent puts the pair in the parentToChild map.
- When refToCloneConf(origConf) is called, ConfAgent firsts clones origConf into a new object newConf. ConfAgent then puts newConf in nodeTable with the nodeID retrieved from threadContext, and puts origConf in unitTestConfIDs (Rule 2). Furthermore, ConfAgent recursively searches origConf's parent in the parentToChild map to move them from uncertainConfIDs to unitTestConfIDs (Rule 3). Finally, ConfAgent returns newConf.

As mentioned previously, if uncertainConfIDs is not empty at the end of a unit test during the pre-run, meaning that ConfAgent cannot properly map certain objects to a node, ConfAgent excludes the test instances that combine this unit test and any parameters used by the configuration objects in uncertainConfIDs from further testing.

When interceptGet(conf, paraName) is called, ConfAgent first searches whether conf is in nodeTable: if so, ConfAgent can retrieve its corresponding nodeType and nodeIndex and check whether TestGenerator has assigned a particular value to <nodeType, nodeIndex, paraName>, in which case interceptGet returns the assigned value. If conf is not in nodeTable or if TestGenerator has not assigned a particular value, ConfAgent returns the original value in conf.

ConfAgent utilizes interceptSet to solve the following problem: sometimes the unit test creates a configuration object with empty values, then creates a node with this configuration object, expecting the node to fill the empty values, and later retrieves these values (e.g., line 8 in Figure 2d). Since we replace the reference with a clone, the unit test is unable to get the correct values after node initialization. To solve this problem, ConfAgent adds the following logic to interceptSet(conf, paraName, paraValue): if conf belongs to a node in nodeTable and the parentConfID of the particular node is not empty, then ConfAgent updates the corresponding value of the parent conf as well.

To illustrate the whole workflow, we next present how ConfAgent works on the example shown in Figure 2.

Step 1: When the unit test starts, it creates a new blank configuration (line 2 in Figure 2d), which triggers ConfAgent's newConf function at line 3 of Figure 2a. At this moment, since no node has been initialized yet (i.e., nodeTable is empty), ConfAgent marks this configuration object as belonging to the unit test (Rule 1.2).

Step 2: The unit test creates server1: the constructor of the Server class triggers ConfAgent's startInit function (line 14 in Figure 2b). This function generates a nodeID for this Server object and then registers a new node with type "Server" and its nodeID in the nodeTable; it also registers nodeID (i.e., server1) in threadContext, indicating that the code of server1 is running on the main thread.

Step 3: The Server constructor triggers the refToClone function of ConfAgent (line 17 in Figure 2b). This function first clones the object. Then it tries to assign the cloned configuration object to a node by searching the threadContext to find which node is running on the current thread. In our example, it finds server1 is running, so it marks the cloned configuration object as belonging to server1 (Rule 2). It also marks the configuration object to be cloned as belonging to the unit test, but since that object was already marked in Step 1, this step does not change anything.

Step 4: The Server constructor creates a component object, which creates its own blank configuration (line 5 in Figure 2c). It triggers ConfAgent's newConf function. In this case, since ConfAgent finds server1 is running on the current thread from threadContext, it marks the new configuration object as belonging to server1.

Step 5: The `Server` constructor triggers ConfAgent's `stopInit` function (line 21 in Figure 2b), which unregisters `server1` from the `threadContext`.

Step 6: The unit test creates `server2` (line 5 in Figure 2d), which repeats Steps 2–5 and marks the corresponding configuration objects as belonging to `server2`.

Step 7: When the unit test calls `funA` (line 7 in Figure 2d), if this function calls `Configuration.get`, ConfAgent can intercept this function (line 17 in Figure 2a) and know that the configuration object belongs to `server1`. ConfAgent can manipulate the return value here to allow `server1` and other nodes to have different configuration values.

6.4 Assumptions

Although we have tried to make our implementation generalize to different applications, it does rely on a few assumptions: 1) the application should have whole-system unit tests; 2) the application should have a well-defined configuration class, which contains all the configuration parameters used by the application; 3) for each type of node, a well-defined initialization function will initialize all its configuration objects, either by creating a new configuration object or by storing a reference to an argument of the function; 4) configuration objects are not stored as global variables, because that would prevent ConfAgent from classifying configuration objects; and 5) different nodes should not share any object that needs to read configuration values, since we cannot determine what value to give to a shared object.

A violation of assumption 2, 3, 4, or 5 does not completely prevent ConfAgent from working: we can always modify the source code to handle these violations, or skip certain parameters if they are shared. Of course the more violations the application has, the less effective ZebraConf becomes. In the evaluation section, we discuss our experience with violations of these assumptions in real-world applications.

7 Experimental Evaluation

We implemented ConfAgent with 641 lines of Java code, TestRunner with 610 lines of Java Code and 1,285 lines of shell script, and TestGenerator with 133 lines of Java code and 327 lines of shell script. We also have 227 lines of shell script to run tests with Docker containers [34].

Our evaluation tries to answer two questions:

- How many heterogeneous-unsafe configuration parameters can ZebraConf find in real-world applications? (§7.1)
- How can the individual techniques of ZebraConf help to improve its accuracy and reduce its running time? (§7.2)

To answer these questions, we have applied ZebraConf to Flink, HBase, HDFS, MapReduce, and YARN. We manually analyzed all the reported problems to understand whether they are true problems or false positives. Our principles for separating true problems and false positives are as follows. 1) To check whether a failed unit test may happen in a real distributed setting to be a true problem, we check two properties: first, a client should not need to manipulate the private data of a server, which is only possible in a unit test, not in a real distributed setting; second, the error should not be caused by an inconsistent configuration within one node, which will not happen in a real distributed setting; 2) If the failed test causes an error in the application code, we classify it as a real problem. 3) If the failed test does not cause any errors in the application code, but violates some assertion in the unit test, we try to understand the assertions and make a best-effort determination whether the assertion would be meaningful in a realistic setting: if yes, we classify it as a real problem; otherwise, we classify it as a false positive.

Testbed. We run all experiments on CloudLab [9]. Each machine is equipped with two Intel Xeon 10-core CPUs, 192 GB DRAM, 480 GB SATA SSD (where we run experiments), and 1 TB SAS HD. We use up to 100 physical machines and allocate 20 Docker containers on each physical machine to run unit tests in parallel.

7.1 Heterogeneous-Unsafe Configuration Parameters in Real-World Applications

ZebraConf reports a total number of 57 heterogeneous-unsafe parameters in the five target applications, and our manual analysis reveals 41 of them are true problems. We list all true problems in Table 3.

We categorize the true problems as follows, and we discuss them in the contexts of both long-term heterogeneous configuration and short-term heterogeneous configuration (i.e., partial reboot in a homogeneous system).

- Compression-, encryption-, authentication-, or transport-protocols-related parameters. These parameters affect the data format in a file or in a network communication, and thus if two nodes have different parameter values, one node will not be able to read data correctly. For a pair of nodes transferring data to each other, there is no reason to use heterogeneous values for these parameters in the long term. Reconfiguring these nodes may create a heterogeneous configuration in the short term, and a possible solution is to store the parameter value for each file or communication channel, so that a reconfiguration will not affect files or channels created before reconfiguration.
- Heartbeat-related parameters. If a heartbeat sender has a large interval value but the receiver has a small value, the sender may not send heartbeats in time, so that the receiver may decide the sender has died. While there is no good reason to use heterogeneous heartbeat intervals in the long term, it may happen in the short term due to the demand to reconfigure such values at run time. For example, since version 2.9.0, HDFS has supported reconfiguring `dfs.heartbeat.interval` at run time with its reconfiguration interface `hdfs dfsadmin -reconfig namenode` [17].

Parameter	Why parameter is heterogeneous unsafe
Flink	
akka.ssl.enabled	TaskManager fails to connect to ResourceManager.
taskmanager.data.ssl.enabled	TaskManager fails to decode peer message due to invalid SSL/TLS record.
taskmanager.numberOfTaskSlots	JobManager fails to allocate slot from TaskManager.
Hadoop Common	
hadoop.rpc.protection	RPC client fails to connect to RPC servers.
ipc.client.rpc-timeout.ms	Socket connection timeouts.
HBase	
hbase.regionserver.thrift.compact	Thrift Admin fails to communicate with Thrift Server.
hbase.regionserver.thrift.framed	Thrift Admin fails to communicate with Thrift Server.
HDFS	
dfs.block.access.token.enable	DataNode fails to register block pools.
dfs.bytes-per-checksum	Checksum verification fails on DataNode.
dfs.blockreport.incremental.intervalMsec	End users may observe inconsistent number of blocks.
dfs.checksum.type	Checksum verification fails on DataNode.
dfs.client.block.write.replace-datanode-on-failure.enable	NameNode reports Exception when Client tries to find additional DataNode.
dfs.client.socket-timeout	Socket connection timeouts.
dfs.datanode.balance.bandwidthPerSec	Balancer timeouts because DataNode fails to reply in time.
dfs.datanode.balance.max.concurrent.moves	Balancer becomes 10x slower due to DataNode congestion control.
dfs.datanode.du.reserved	End users may observe inconsistent size of reserved space.
dfs.data.transfer.protection	Sasl handshake fails between Client and DataNode.
dfs.encrypt.data.transfer	DataNode fails to re-compute encryption key as block key is missing.
dfs.ha.tail-edits.in-progress	JournalNode declines NameNode's request to fetch journaled edits.
dfs.heartbeat.interval	NameNode falsely identifies alive DataNode as crashed.
dfs.http.policy	Tool DFSck fails to connect to HTTP server.
dfs.namenode.fs-limits.max-component-length	Length of component name path exceeds maximum limit on NameNode.
dfs.namenode.fs-limits.max-directory-items	Directory item number exceeds maximum limit on NameNode.
dfs.namenode.heartbeat.recheck-interval	End users may observe inconsistent number of dead DataNodes.
dfs.namenode.max-corrupt-file-blocks-returned	End users may observe inconsistent number of corrupted blocks.
dfs.namenode.snapshotdiff.allow.snap-root-descendant	NameNode declines Client's request to do snapshot.
dfs.namenode.stale.datanode.interval	End users may observe inconsistent number of stale DataNodes.
dfs.namenode.upgrade.domain.factor	Balancer hangs because of block placement policy violation on NameNode.
MapReduce	
mapreduce.fileoutputcommitter.algorithm.version	Different Mapper/Reducer output commit dirs cause Hadoop Archive error.
mapreduce.job.encrypted-intermediate-data	Reducer fails during shuffling due to checksum error.
mapreduce.job.maps	Reducer fails when copying Mapper output.
mapreduce.job.reduces	Reducer fails when copying Mapper output.
mapreduce.map.output.compress	Reducer fails during shuffling due to incorrect header.
mapreduce.map.output.compress.codec	Reducer fails during shuffling due to incorrect header.
mapreduce.output.fileoutputformat.compress	End users may observe inconsistent names of output files.
mapreduce.shuffle.ssl.enabled	NodeManager's Pluggable Shuffle fails to decode messages.
Yarn	
yarn.http.policy	Client fails to connect with Timeline web services.
yarn.resourcemanager.delegation.token.renew-interval	End users may observe newer tokens expire earlier than prior tokens.
yarn.scheduler.maximum-allocation-mb	ResourceManager disallows value decreasement.
yarn.scheduler.maximum-allocation-vcores	ResourceManager disallows value decreasement.
yarn.timeline-service.enabled	Client fails to connect to Timeline Server.

Table 3. The 41 true heterogeneous-unsafe configuration parameters found by ZebraConf.

Such an online reconfiguration will create a short-term invalid heterogeneous configuration.

We propose the following workaround: if the administrator needs to decrease the heartbeat interval, she should change the value at the heartbeat sender first, and then change the value at the receiver; if the administrator needs to increase the interval, she should change it at the receiver first and then at the sender. This strategy ensures that the sender interval is always less than or equal to the receiver interval, so that the receiver will not miss heartbeats. However, this

workaround may not always work, since sometimes a node can act as both the sender and the receiver.

- Max-limit-related parameters. These parameters can encounter problems if we reconfigure a node's max limit to be smaller, while the state of the node already exceeds the smaller limit. The administrator should simply not try to reconfigure a node to decrease the max limit. In contrast, increasing the limit causes no problems in our experiments.
- Counts of tasks. Nodes with inconsistent values of these parameters will have problems retrieving data. These parameters should not be configured in a heterogeneous manner.
- Others. This group contains some interesting parameters that we did not expect to be unsafe. We provide details about some of them as follows.

dfs.datanode.balance.bandwidthPerSec. This parameter is used to specify the maximum amount of bandwidth that each HDFS DataNode can utilize for balancing. Starting in HDFS 0.20, developers made this parameter online reconfigurable with a new dfsadmin command, because administrators found "the optimal value of the bandwidthPerSec parameter is not always (almost never) known at the time of cluster startup" [18]. Setting `bandwidthPerSec` either too low or too high may bring the cluster into a "maintenance window," which is expensive for large clusters. Making this parameter reconfigurable at run time would help to avoid or alleviate this issue.

However, when setting this parameter in a heterogeneous manner, we observe the following problem: a DataNode with a high bandwidth limit may send many packets to a DataNode with a low limit so that the latter may run out of its quota. In this case, the latter will throttle its network traffic, which is expected. However, such throttling may prevent the DataNode from sending progress report to the Balancer, which is a tool to balance data in the cluster. As a result, the Balancer times out eventually. To solve this problem, we propose that each node should reserve a small fraction of bandwidth for critical traffic like heartbeats or progress reports.

dfs.datanode.balance.max.concurrent.moves. A unit test reports timeout (100 s) when this parameter is configured to be 1 on DataNodes and 50 on the Balancer. The default value for this parameter is 50, which allows 50 threads on a DataNode to transfer blocks for balancing.

We checked the average balancing time for several configurations in the unit test: the time for (DataNode:50, Balancer:50) was 14 seconds and for (DataNode:1, Balancer:1) was 16.7 seconds—but the time for (DataNode:1, Balancer:50) was 154 seconds. While it made sense that (DataNode:50, Balancer:50) was faster than (DataNode:1, Balancer:1), it was initially unclear why (DataNode:1, Balancer:50) was significantly slower than (DataNode:1, Balancer:1).

Our further investigation showed that in (DataNode:1, Balancer:50), because Balancer is unaware of the 1-thread capacity on DataNodes, it still sends block transfer requests to DataNodes concurrently. However, a DataNode declines requests when its thread is already performing block transfer for balancing. When the request is declined, the corresponding Balancer dispatcher thread triggers a congestion control mechanism, which sleeps for 1100 ms before it retries. Since the DataNodes usually can finish usually can finish a block transfer request within 1100 ms, such congestion control adds an extra delay to the whole procedure.

The reader may wonder why we want to set this parameter differently on Balancer and on DataNodes in the first place. Indeed, if all DataNodes have the same value, there is no good reason for the Balancer to use a different value. However, if different DataNodes have different values, then it is inevitable that the Balancer's configuration will be different from some of the DataNodes. Because of the reported problem, it seems like a bad idea for the Balancer to use one value for this parameter. Instead, the Balancer should retrieve this value from different DataNodes, and accordingly send different numbers of tasks to different DataNodes. We observe that the community is already discussing this solution [19].

dfs.namenode.upgrade.domain.factor. This parameter is in effect when HDFS's block placement policy is set to `BlockPlacementPolicyWithUpgradeDomain`. *Upgrade domain* is a feature to support rolling upgrade, which upgrades a subset of DataNodes at a time. To minimize the chance of data unavailability, a rolling upgrade should affect at most one replica of a data block at a time. To satisfy this property, HDFS allows the administrator to divide DataNodes into groups, called upgrade domains, and HDFS ensures the replicas of a data block are placed into different upgrade domains.

A unit test that tests whether data rebalancing still honors a domain-aware block placement policy fails when Balancer and NameNode are configured with different numbers of upgrade domains. The rebalancing task never finishes because some block transfer requests are always declined by NameNode, which identifies the block transfer as an action that results in a violation of the placement policy being used.

Similar to the previous problem, if different NameNodes[1] have the same number of UpgradeDomains, there is no good reason for the Balancer to use a different value. If different NameNodes have different numbers of UpgradeDomains, however, it is inevitable for Balancer's configuration to be different from some HDFS NameNode's configuration. A possible solution for this issue is to let Balancer fetch the value of the domain factor from the corresponding NameNode, instead of reading from its local configuration file.

dfs.blockreport.incremental.intervalMsec. This parameter determines whether, if a DataNode deletes a data block, the NameNode will receive the update immediately or whether the update can be delayed. If the DataNode is configured to use the delayed mode and the client's configuration file says a block deletion is reported immediately, a user may issue a

[1] Multiple NameNodes have been supported by HDFS since version 0.23.

delete command, expecting the block to disappear immediately, yet later find the block is still present (HDFS has one unit test simulating this case).

It is debatable whether this parameter presents a true problem. On the one hand, it does not cause any explicit errors in the application; on the other hand, it does expose an inconsistency to the user since the application's behavior does not match the configuration value. We find a total of 16 parameters having similar problems in our study. Our principle for separating true problems and false positives is that if the user can observe an inconsistency through the application's public APIs, then we mark the corresponding parameter as a true problem. If an inconsistency can only be observed through the application's private functions, we mark it as a false positive. For the 16 parameters that cause similar issues, this principle separates them into 7 true problems and 9 false positives.

Such problems show that it is often risky for an application user to make assumptions about the internal implementation of the application, and thus it may be better not to expose internals-related parameters to the application users.

Causes of false positives. We summarize the top causes of false positives as follows:

- The setting does not happen in a real distributed system. Some tests check or manipulate the private data of a node, which cannot happen in a real system. For example, an HBase test directly opens a new region on HRegionServer by calling `HRegionServer.openRegion`, with the client's configuration object. In a real distributed setting, an HBase client can only do so through an RPC, in which case the server will use its own configuration object.
- Violating assumptions. In the unit tests of Hadoop projects, different nodes share the InterProcess Communication (IPC) component, which has its own configuration object. However, the IPC component sometimes reads configuration values from external configuration objects as well. The combination of sharing the IPC component and the IPC component reading values from different places causes the IPC component to read different values in a heterogeneous test, which leads to false alarms for four IPC-related configuration parameters. After we modified one line of code in Hadoop to disable the sharing, the false alarms disappeared.
- Overly strict assertions. Many unit tests use assertions to check the state of the target application. While many of them are meaningful and reveal real problems, we find that a few are overly strict. For example, one test compares the image files of different NameNodes to ensure they are the same, which is meaningful, but it first unnecessarily compares the lengths of the two files. In a heterogeneous setting in which one NameNode compresses the image but the other NameNode does not, their image file lengths are different but their actual contents are still the same.

In our experiments, we find that false positives are usually not hard to identify, since unrealistic settings and strict assertions are usually explicit in the unit test code, which is usually short and easy to understand. The one exception is false alarms caused by IPC sharing, which took us one day to figure out. Understanding the reasons for true problems, on the other hand, is a lot harder, since they often require a deep understanding of how the whole system works.

Application	Modified LOC
Flink	30 + 8
Hadoop Common	0 + 6
HBase	16 + 7
HDFS	24 + 6
MapReduce	12 + 6
Yarn	12 + 6

Table 4. Modified lines of code to apply ZebraConf to each application. The first number is lines related to modifying the node classes, and the second number is lines related to modifying the configuration class.

7.2 Effects of Individual Techniques

Effort to modify the applications. As discussed in Section 6.3, to use ZebraConf, the user needs to use ConfAgent's API to modify two types of class files: the node class and the configuration class. Table 4 shows the lines of code we needed to modify in each application. As the table shows, the modifications to support ZebraConf required low effort.

Among these applications, we find all applications have well-defined configuration objects. HDFS, HBase, MapReduce, and YARN have well-defined initialization functions for each type of node. Flink is more complicated: its node class has initialization functions, which are used in a real distributed setting, but its unit tests do not invoke the initialization functions directly and instead copy the initialization code into the unit test code. It is unclear to us why Flink uses such a design; in any case, it required additional effort on our part to identify and annotate the copied initialization code.

Reduction of number of tests to run. Table 5 presents the effects of individual techniques incorporated by ZebraConf.

The first row is the number of test instances ZebraConf would run assuming the user has the same level of expertise as us but does not pre-run those unit tests as ZebraConf does. In particular, it assumes that the user tests each parameter independently, selects parameter values in the same way as us, and selects value assignment in the same way as us (Section 4). It also assumes the user knows which types of nodes the corresponding application includes, so she will not test nodes or parameters not included in the application. For example, for HDFS, she will not test RegionServer and related parameters, which belong to HBase; for HBase, however, she will test HDFS NameNode or DataNode and related parameters, because HBase depends on HDFS. As one can see in the table, even with these strategies, ZebraConf needs to run a large number of tests.

	Flink	Hadoop-Tools	HBase	HDFS	MapReduce	YARN
Original	7,193,881,080	373,850,400	557,761,680	387,499,008	284,486,160	705,346,824
After pre-running unit tests	2,019,422	356,016	6,145,374	10,404,952	482,272	668,020
After removing uncertainty	1,972,278	346,588	6,033,174	10,242,886	430,800	640,338
After pooled testing	259,573	89,744	1,438,929	1,968,218	104,588	312,726

Table 5. The number of test instances generated after successively applied methods.

The second row reports the number of test instances after we pre-run the unit tests to filter ineffective ones. By looking at the test information, we observe that 1) many unit tests, which are designed to test individual data structures, do not even start any nodes, and thus are completely filtered by this step; 2) almost no unit tests use all parameters; 3) even for unit tests that use a certain parameter, in many cases the parameter is only used by a subset of nodes. Together these reasons allow us to reduce the number of test instances to run by up to three orders by magnitude.

The third row computes the number of test instances after we remove those with uncertain configuration objects (Section 6). As one can see, most of the test instances do not encounter uncertain configuration objects, confirming the accuracy of ZebraConf's approach to map configuration objects to nodes. Note however that although the percentage of tests with uncertainties is small, if we did not remove them, they would yield a high false positive rate, because the percentage of unsafe parameters among all parameters is small as well.

The last row records the number of test instances ZebraConf actually runs after applying pooled testing, including both the executed pooled tests and individual tests (when a pooled test fails). As the table shows, pooled testing further reduces the number of tests ZebraConf needs to run.

These techniques reduce the number of tests to run by two to four orders of magnitude. With their help, ZebraConf is able to finish all tests within 4,652 machine hours. While this number is not small, it is affordable considering that an application does not need to be tested by ZebraConf frequently.

Effects of hypothesis testing. In our experiments, ZebraConf reported 2,167 test instances as failed in the first trial (i.e., the heterogeneous configuration test failed but all corresponding homogeneous tests succeeded), and hypothesis testing filtered 731 of the tests as false positives. These numbers have confirmed the necessity of hypothesis testing.

7.3 Lessons Learned

Our overall conclusion is that, to support heterogeneous configuration in a large system, we need to rethink how configuration should be used and tested. First, the existing paradigm, in which each node reads configuration values from its configuration file, is not sufficient anymore, since a node may need to communicate with nodes with different configurations. Instead, a node may need to ask for configuration values from other nodes. Therefore, it may be necessary to split parameters that are local to a node and those that must be agreed upon by different nodes. Embedding parameter values in the communication or in the file, instead of relying on configuration files, may be a good practice to solve this problem. Applying these techniques would prevent about 20 unsafe parameters in Table 3, including *dfs.datanode.balance.max.concurrent.moves*, *dfs.namenode.upgrade.domain.factor*, and those related to compression, encryption, and similar.

Second, we find that several problems are caused by exposing unnecessary parameters to end users, since these parameters may serve as a side channel for the end users to rely on the implementation details of the application. Following general software engineering principles, an end user should not be allowed to read the value of a parameter if the parameter is solely used internally by the application. This rule would prevent seven parameters from causing inconsistencies between the end user and the application as listed in Table 3.

Finally, whole-system unit tests provide a convenient and efficient solution compared to integration tests, but to support them, both the application code and the unit test code should follow a style that is friendly to whole-system unit tests, e.g., they should not use global variables or share objects across different nodes. These changes will not only reduce false negatives and false positives in ZebraConf, but also make those unit tests more closely resemble a real distributed setting.

8 Conclusion

ZebraConf is a framework for utilizing existing unit tests to identify heterogeneous-unsafe configuration parameters. ZebraConf provides an API to modify the target application with minimal effort and a number of strategies to reduce the number of tests to run. Our evaluation on five popular open-source applications finds 41 heterogeneous-unsafe configuration parameters, which implies that an administrator should be careful when applying heterogeneous configurations. The source code of ZebraConf is publicly available: https://github.com/StarThinking/ZebraConf/.

Acknowledgments

We thank all reviewers for their insightful comments, especially our shepherd Pedro Fonseca for his guidance during camera-ready preparation. This material is based in part upon work supported by the National Science Foundation under Grant Numbers CNS-1908020 and XPS-1629126.

References

[1] Apache Flink. https://flink.apache.org.

[2] Apache HBASE. http://hbase.apache.org.

[3] Mona Attariyan and Jason Flinn. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, Vancouver, BC, Canada, October 2010.

[4] Liang Bao, Xin Liu, Ziheng Xu, and Baoyin Fang. Autoconfig: Automatic configuration tuning for distributed message systems. In *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE'18)*, Montpellier, France, September 2018.

[5] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, et al. Apache Hadoop Goes Realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD'11)*, Athens, Greece, June 2011.

[6] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems. In *2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*, Virtual Event, November 2020.

[7] Dazhao Cheng, Jia Rao, Yanfei Guo, Changjun Jiang, and Xiaobo Zhou. Improving Performance of Heterogeneous MapReduce Clusters with Adaptive Task Tuning. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):774–786, 2016.

[8] Cloudera Documentation - Rolling Restart. https://docs.cloudera.com/documentation/enterprise/6/6.3/topics/cm_mc_rolling_restart.html.

[9] CloudLab. https://www.cloudlab.us.

[10] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning Database Configuration Parameters with iTuned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.

[11] Apache Flink Documentation: Configuration. https://ci.apache.org/projects/flink/flink-docs-stable/deployment/config.html.

[12] Allow Configuration Changes without Restarting Configured Nodes. https://issues.apache.org/jira/browse/HADOOP-7001.

[13] Hadoop Hadoop Common Configuration File. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/core-default.xml.

[14] HBase-8544. Add a Utility to Reload Configurations in Region Server. https://issues.apache.org/jira/browse/HBASE-8544.

[15] HBase Configuration File. https://hbase.apache.org/book.html#hbase_default_configurations.

[16] Hadoop HDFS Project. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[17] HDFS-1477. Support Reconfiguring 'dfs.heartbeat.interval' and dfs.namenode.heartbeat.recheck-interval without NN Restart. https://issues.apache.org/jira/browse/HDFS-1477.

[18] HDFS-2202. Changes to Balancer Bandwidth Should Not Require Datanode Restart. https://issues.apache.org/jira/browse/HDFS-2202.

[19] HDFS-7466. Allow Different Values for 'dfs.datanode.balance.max.concurrent.moves' per Datanode. https://issues.apache.org/jira/browse/HDFS-7466.

[20] Hadoop HDFS Configuration File. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml.

[21] Hadoop HDFS RBF Configuration File. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs-rbf/hdfs-rbf-default.xml.

[22] Herodotos Herodotou and Shivnath Babu. Profiling, What-If Analysis, and Cost-based Optimization of MapReduce Programs. *Proceedings of the VLDB Endowment*, 4(11):1111–1122, 2011.

[23] Peng Huang, William J. Bolosky, Abhishek Singh, and Yuanyuan Zhou. ConfValley: A Systematic Configuration Validation Framework for Cloud Services. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*, Bordeaux, France, April 2015.

[24] Apache Kafka. https://kafka.apache.org.

[25] Confluent Documentation: Dynamic Configurations. https://docs.confluent.io/platform/current/kafka/dynamic-config.html.

[26] Palden Lama and Xiaobo Zhou. AROMA: Automated Resource Allocation and Configuration of Mapreduce Environment in the Cloud. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC'12)*, San Jose, California, USA, September 2012.

[27] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R. Butt, and Nicholas Fuller. MRONLINE: MapReduce Online Performance Tuning. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC'14)*, Vancouver, BC, Canada, 2014.

[28] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. OPTIMUSCLOUD: Heterogeneous Configuration Optimization for Distributed Databases in the Cloud. In *2020 USENIX Annual Technical Conference (USENIX ATC'20)*, Virtual Event, July 2020.

[29] Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh, Subrata Mitra, Wolfgang Gerlach, Travis Harrison, Folker Meyer, Ananth Grama, Saurabh Bagchi, and Somali Chaterji. Rafiki: A Middleware for Parameter Tuning of NoSQL Datastores for Dynamic Metagenomics Workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware'17)*, Las Vegas, Nevada, USA, December 2017.

[30] Ashraf Mahgoub, Paul Wood, Alexander Medoff, Subrata Mitra, Folker Meyer, Somali Chaterji, and Saurabh Bagchi. SOPHIA: Online Reconfiguration of Clustered NoSQL Databases for Time-Varying Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, Renton, WA, USA, July 2019.

[31] Hadoop MapReduce Configuration File. https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml.

[32] Hadoop MapReduce Project. https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[33] MAPREDUCE-442. Ability to Re-configure Hadoop Daemons Online. https://issues.apache.org/jira/browse/MAPREDUCE-442.

[34] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux journal*, 2014(239):2, 2014.

[35] MySQL 8.0 Reference Manual - Performing a Rolling Restart of an NDB Cluster. https://dev.mysql.com/doc/refman/8.0/en/mysql-cluster-rolling-restart.html.

[36] Ariel Shemaiah Rabkin. *Using Program Analysis to Reduce Misconfiguration in Open Source Systems Software*. PhD thesis, UC Berkeley, 2012.

[37] Redis. https://redis.io.

[38] Redis Commands: CONFIG SET Parameter Value. https://redis.io/commands/config-set.

[39] Daniel Sun, Alan Fekete, Vincent Gramoli, Guoqiang Li, Xiwei Xu, and Liming Zhu. R2C: Robust Rolling-Upgrade in Clouds. *IEEE Transactions on Dependable and Secure Computing*, 15(5):811–823, 2016.

[40] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. Testing Configuration Changes in Context to Prevent Production Failures. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Virtual Event, November 2020.

[41] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. Understanding and Auto-Adjusting

Performance-Sensitive Configurations. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*, Williamsburg, VA, USA, March 2018.

[42] Chengcheng Xiang, Haochen Huang, Andrew Yoo, Yuanyuan Zhou, and Shankar Pasupathy. PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations. In *2020 USENIX Annual Technical Conference (USENIX ATC'20)*, pages 265–280. USENIX Association, July 2020.

[43] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Savannah, GA, USA, nov 2016.

[44] Hadoop YARN Project. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[45] Hadoop YARN Configuration File. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-common/yarn-default.xml.

[46] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. Context-based Online Configuration-Error Detection. In *2011 USENIX Annual Technical Conference (USENIX ATC'11)*, Portland, OR, USA, June 2011.

[47] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*, Amsterdam, Netherlands, June 2019.

[48] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, Salt Lake City, Utah, USA, March 2014.

[49] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. The Inflection Point Hypothesis: a Principled Debugging Approach for Locating the Root Cause of a Failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, Huntsville, Ontario, Canada, October 2019.

[50] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*, Santa Clara, California, September 2017.