Systems Approaches to Tackling Configuration Errors: A Survey

TIANYIN XU and YUANYUAN ZHOU, University of California San Diego

In recent years, configuration errors (i.e., misconfigurations) have become one of the dominant causes of system failures, resulting in many severe service outages and downtime. Unfortunately, it is notoriously difficult for system users (e.g., administrators and operators) to prevent, detect, and troubleshoot configuration errors due to the complexity of the configurations as well as the systems under configuration. As a result, the cost of resolving configuration errors is often tremendous from the aspects of both compensating the service disruptions and diagnosing, recovering from the failures. The prevalence, severity, and cost have made configuration errors one of the most thorny system problems that desire to be addressed.

This survey article provides a holistic and structured overview of the systems approaches that tackle configuration errors. To understand the problem fundamentally, we first discuss the characteristics of configuration errors and the challenges of tackling such errors. Then, we discuss the state-of-the-art systems approaches that address different types of configuration errors in different scenarios. Our primary goal is to equip the stakeholder with a better understanding of configuration errors and the potential solutions for resolving configuration errors in the spectrum of system development and management. To inspire follow-up research, we further discuss the open problems with regard to system configuration. To the best of our knowledge, this is the first survey on the topic of tackling configuration errors.

Categories and Subject Descriptors: D.4.5 [**Operating Systems**]: Reliability; C.5 [**Computer System Implementation**]; D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.9 [**Software Engineering**]: Management

General Terms: Design, Reliability, Configuration

Additional Key Words and Phrases: Configuration, misconfiguration, configuration error, failure, automation, testing, vulnerability, detection, validation, deployment, management, diagnosis, troubleshooting

ACM Reference Format:
Tianyin Xu and Yuanyuan Zhou. 2015. Systems approaches to tackling configuration errors: A survey. ACM Comput. Surv. 47, 4, Article 70 (July 2015), 41 pages.
DOI: http://dx.doi.org/10.1145/2791577

1. INTRODUCTION

Configuration errors (i.e., misconfigurations) have become one of the major causes of today's system failures. For example, Barroso and Hölzle [2009] report that configuration errors were the second major cause of the service-level failures at one of Google's main services. Rabkin and Katz [2013] report that configuration errors were the dominant cause of Hadoop cluster failures, in terms of both the number of customer cases and the supporting time. Similar statistics have been observed in other types of systems and applications, such as storage systems [Jiang et al. 2009; Yin et al. 2011], data-intensive systems [Yuan et al. 2014], large-scale Internet and Web services [Oppenheimer et al. 2003], cloud systems [Gunawi et al. 2014], database servers [Oliveira et al. 2006], and

Authors' addresses: T. Xu and Y. Zhou, Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Drive, La Jolla, CA 92093; emails: {tixu, yyzhou}@cs.ucsd.edu.
Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2015 ACM 0360-0300/2015/07-ART70 $15.00
DOI: http://dx.doi.org/10.1145/2791577

backbone networks [Labovitz et al. 1999]. In recent years, we have witnessed that a number of large Internet and cloud-service providers (e.g., Google, Microsoft, Amazon, LinkedIn) experienced misconfiguration-induced outages, affecting millions of their customers [Thomas 2011; Brodkin 2012; Miller 2012; Liang 2013].

Unfortunately, it is notoriously difficult to prevent, detect, and troubleshoot configuration errors due to the complexity of the configurations as well as the systems under configuration. The complexity of configurations is reflected not only by the large number of configuration parameters, but also by their correlations and dependencies (both inside the software programs and across different software components). Typically, a large, complex software system exposes hundreds of configuration parameters, and each parameter has its setting constraints (i.e., correctness rules or requirements) [Kiciman and Wang 2004; Nadi et al. 2014]; many configuration constraints are neither intuitively designed nor well documented [Rabkin and Katz 2011b; Xu et al. 2013]. Moreover, many configuration parameters have correlations and dependencies with other parameters or the system's runtime environment [Benson et al. 2011; Zhang et al. 2014], which further increases the complexity. Such complexity poses significant challenges for users (e.g., system administrators and operators) to correctly perform configuration tasks, and for system vendors to troubleshoot users' misconfigurations. Without advanced systems approaches, tackling configuration errors by manual processes of trial and error or examining the entire error space is obviously inefficient and time consuming. As it is hard to locate the configuration error "in the haystack," users often cannot fix the errors and have to report their problems to the vendors, resulting in expensive diagnosis cycles and long recovery time.

In fact, configuration errors incur high costs not only because of the urgency of resolving the errors and recovering the disrupted services, but also because they escalate the supporting cost of system vendors who are responsible for helping users troubleshoot and correct configuration errors. A recent study [Yin et al. 2011] shows that configuration-related issues account for 27% of all the customer-support cases in a major storage company in the U.S., and are the most significant contributor (31%) among all the high-severity support cases. As estimated by Computing Research Association [2003], administrative expenses are made up almost entirely of people costs, representing 60%–80% of the total cost of IT ownership. Of these administrative expenses, system configuration is one of the major operations.

With their prevalence, severity, and cost, configuration errors have been treated as one of the most important yet thorny system problems. Welsh [2013] ranked "*an escape from configuration hell*" as the number one problem among all that he wished system researchers would work on in 2013. In Google Faculty Summit 2011, Wilkes [2011] named configuration as the next big challenge for the system research. In the context of cloud systems, Gunawi et al. [2014] call for research in the cloud community to deal with configuration issues.

In fact, a number of research and practical approaches have been proposed to tackle configuration errors. These approaches span across a broad spectrum of system development and management processes. For example, automated approaches have been developed for a number of configuration tasks, in order to prevent users from misconfiguring the systems in the first place. To help developers harden systems against configuration errors, advanced testing approaches are proposed to expose the system's vulnerabilities to configuration errors so that the developers can fix them proactively. Moreover, there are detection approaches to help system administrators identify potential errors before deploying and rolling out the configuration settings onto the production systems. In the presence of misconfiguration-induced failures, troubleshooting methods are proposed to facilitate support engineers in finding the root causes based on failure-site information.

As the *first* survey on the topic of dealing with configuration errors, this article provides a holistic and structured overview of the existing systems approaches that tackle configuration errors. In order to understand the problem fundamentally, we start with the discussion of the characteristics of configuration errors and the challenges of tackling configuration errors. Then, we discuss the state-of-the-art systems approaches that address different types of configuration errors in different scenarios. Our primary goal is to equip the stakeholder with a better understanding of configuration errors and the potential solutions for resolving configuration errors in the spectrum of system development and management. To inspire follow-up research, we also discuss the open problems with regard to system configuration. We believe that the breakthrough of these problems will lead to the steps toward solving the configuration problems.

In this article, we discuss the configuration errors from the system's perspective and mainly focus on configuration errors in the context of operating systems, server systems, and infrastructure systems that construct the supporting platforms for end-user applications and services. The configurations of these systems are more complex and error prone, compared with end-user applications (e.g., desktop and mobile applications). Most importantly, configuration errors of these systems have a large impact and may affect all the applications and services running on top of them.[1]

2. PRELIMINARY: CONFIGURATION AND CONFIGURATION ERRORS

2.1. What is Configuration?

We define *configuration* to be the operations that determine the values of system settings. The goal of configuration is to transform the system from an unsatisfying state to the intended state by changing the settings. Here, the *state* of a system can be measured by a few aspects of system behavior, such as functionality, performance, reliability, security, etc.

Configuration belongs to system management operations.[2] It has attracted significant attention among all the management operations due to the prevalence and severity of configuration errors, as demonstrated in Section 1. Feamster and Balakrishnan [2005] regard configuration as the most time-consuming and error-prone operations in network management. Oppenheimer et al. [2003] observe that configuration errors formed the largest category of operation errors in the three large Internet services they studied—"*more than 50% (and in one case nearly 100%) of the operator errors that led to service failures were configuration errors.*"

Configuration parameters. Most configurations can be parameterized, and configuration can be viewed as the process of setting *configuration parameters*. To provide users (e.g., administrators and operators) with the flexibility of control, modern software systems often expose a large number of configuration parameters. By setting these parameters, users can control a variety of aspects of system behavior based on their own requirements and preferences, such as

—*environment information*, which is required by runtime execution; for example, the local port to listen on, the remote IP addresses to connect to, and the file-system locations for storing data;

—*functionalities/features selection*; for example, SSL support, debugging support;

[1]The configurations of end-user applications are mainly for the purpose of preference selection and personalization. It has different characteristics and the problems are different from systems that provide operating, server, and infrastructure support.

[2]The other system management operations include hardware management, planning and provisioning (e.g., migrating data to new disks), scripting and programming, etc.

—*resource* provisioning and constraints; for example, memory allocation, limit of CPU consumption, limit of file descriptors, number of disk drives;
—*performance* tuning; for example, synchronization methods, sorting algorithms;
—*security and permission* control on system resources such as Access Control Lists (ACLs);
—*reliability and availability* including system reliability (e.g., standby servers for failover) and data availability (e.g., number of data replicas).
—*versions* specifying particular version(s) of the system components;
—*modules* specifying the loading of particular software modules and libraries.

Note that the configuration settings of one parameter may have correlations or dependencies with the settings of the other parameters. For example, the configuration settings of network services can be logically grouped into *stanzas* [Benson et al. 2011]. The settings inside a stanza together define a functionality, while multiple stanzas have dependencies defined by the settings of multiple stanzas. Zhang et al. [2014] report that 27%–51% of the configuration parameters in their studied open-source software projects have correlations with other parameters.

Also, the settings of a parameter may affect multiple aspects of the system state. For example, increasing the number of data replicas could improve the data availability and performance (e.g., reducing response latency), but results in low resource (storage) utilization.

Storage of configurations. Configuration settings are usually stored in software-specific text files or centralized configuration repositories. The former is adopted by UNIX-like systems, while the latter is exemplified by the Windows Registry (a hierarchical database that stores configuration settings of OS and applications of Microsoft Windows). The persistent storage (files and databases) also serves as the interface of configuration: users are asked to edit the configuration files or Registry entries to configure the system. Some systems provide special configuration User Interfaces (UIs) to assist users during their configuration processes to enforce checking and provide prompt feedback.

2.2. Who Performs Configuration Tasks?

Many software systems need to be configured before the delivery of the expected functionalities or services, and constantly tuned according to the changes of workloads and runtime conditions. Typically, configuration is performed by *system administrators* (also known as *system operators*) who are responsible for operating and managing the systems. Unlike those in many other professions, many system administrators do not come from traditional computer science and engineering backgrounds due to the lack of education programs [Border and Begnum 2014].[3] Wikipedia [2014] describes that "*there is no single path to becoming a system administrator... a system administrator usually has a background of related fields such as computer engineering, information technology, information systems, or even a trade school program... many system administrators enter the field by self-taught....*" The survey conducted by StackOverflow [2015] shows that system administrators are the most likely to be self-taught among all the programmers—52% of the surveyed system administrators learn programming on their own.

System administrators are very different from software developers for the lack of the same level of *understanding*, *controlling*, and *debugging support*. Unlike developers who understand the system programs they write and can control and debug the

[3]Recently, the education of system administrators has attracted more attention: several administrator education groups and programs have been developed [Border and Begnum 2014].

programs, system administrators often do not write the code and usually do not (or cannot) read the code, which impairs their understanding of the systems and the offered configuration parameters. Moreover, system administrators cannot debug the program in the same way as developers (e.g., interactively examining system internal states) when they encounter problems, especially for commercial software.

System administrators are also very different from ordinary computer users, that is, *end users*. They are required to have technical expertise and experience of system operations. The systems they operate are usually significantly larger and more complex than end-user applications. Many of these systems are not designed with novices in mind but assume a certain level of knowledge and background. For example, there is little guidance and feedback offered by the systems for configuration [Hubaux et al. 2012]. Thus, administrators mainly rely on user manuals or Internet resources to understand the systems and perform configuration tasks.

In recent years, due to the proliferation of open-source software and the on-demand cloud computing infrastructure, the system administrator groups have shifted. Today, configuration is not only performed by well-trained professional system administrators from large enterprises and organizations, but also by pluralistic and novice administrators who operate their own systems (e.g., Web services). As a result, some of the early observations and assumptions on professional system administrators (e.g., Hrebec and Stiber [2001], Barrett et al. [2004], Haber and Bailey [2007], and Velasquez et al. [2008]) may no longer hold. We discuss the shift and its impact in more detail in Section 11.

In this article, we use the term "user" to represent a diverse set of people who perform system configuration, including both professional administrators and operators, as well as users who run systems to support their work or hobbies.

2.3. What Are Configuration Errors?

Following the definition of configuration in Section 2.1, *configuration errors* are errors in the system's configuration settings. Configuration errors are often manifested through incorrect system states, such as system failures, performance degradation, and the other unintended system behavior.[4] It is worth noting that configuration errors by definition could be subjective to the requirements and expectations of the users, especially when they do not cause failures (e.g., crashing and hanging the system). For example, the configuration settings that lead to certain degrees of performance degradation could be considered as misconfigurations for the users competing for performance, but be perfectly acceptable for the other users.

Yin et al. [2011] classify software configuration errors into the following three categories:

—*Parameter:* erroneous settings of configuration parameters (either an entry in a configuration file or a console command);
—*Compatibility:* configuration errors related to software incompatibility;
—*Component:* the other remaining configuration errors (e.g., missing a specific software module).

Most of the existing research efforts focus on parameter misconfigurations, because they account for the majority of real-world configuration errors (e.g., 70.0%–85.5% of the studied misconfiguration cases in Yin et al. [2011]). Moreover, as discussed in Section 2.1, the configurations of compatibility and components can also be modeled and represented by parameters. For example, HBase (Hadoop Database) uses the

[4]In this article, we follow the Laprie [1995] definition of failures. A system *failure* means that the delivered service deviates from fulfilling the desired functions, including crashes, hangs, incorrect results, and incomplete functionalities.

`hfile.format.version` parameter to specify the file formats of different versions for compatibility checking; in Apache HTTP server, the `LoadModule` parameter is used for loading executable modules (i.e., component) at startup time.

Configuration errors differ from *software bugs* in certain ways. First, unlike software bugs, configuration errors are not defects inside the software itself (source code).[5] Even perfectly coded software can be misconfigured by the users. As a result, traditional bug detection approaches based on program analysis (either statically or dynamically) cannot deal with configuration errors. Second, different from software bugs that are supposed to be fixed by developers, configuration errors can be fixed directly by users themselves.[6] Third, debugging may not be an option for users (e.g., administrators) by which to troubleshoot configuration errors, mainly because of users' lack of expertise or not having debugging information (e.g., source code), as discussed in Section 2.2.

2.4. What Causes (So Many) Configuration Errors?

Configuration errors can be introduced in many ways. Based on the subject that committed the errors, we classify the causes of configuration errors into system errors and human errors.

System Errors. Configuration errors could be caused by data corruption as a result of system errors. Wang et al. [2004] list a number of such system errors in the context of Windows Registries, such as failed uninstallation of applications, applied patches that are incompatible with existing applications, and software bugs that corrupt the configuration files on the disk.

Human Errors. The other configuration errors are users' misconfigurations, that is, human errors. Some of the misconfigurations are caused by users' inadvertent action *slips* when setting configuration parameters. For example, the large Internet outage in 2002 was caused by the misplacement of a single bracket in a complex route filter rule [Slater 2002]. A typo (missing a dot) in a domain name of DNS configurations dropped Sweden off the Internet for 1.5 hours [McNamara 2009]. Unfortunately, even experienced, well-trained system administrators may slip, especially under stress [Brown 2001].

The majority of misconfigurations are caused by users' *mistakes* in choosing the values of configuration parameters due to misunderstanding, ignorance, and failures of reasoning. Examples of such mistakes are not only when users incorrectly set configuration values, but also when they do not set the parameters that should be configured according to the environments or workloads. This is understandable considering the following characteristics of system configuration.

—*Complexity:* The complexity of configurations is one essential reason for configuration errors. As many of today's systems expose a large number of configuration parameters, it is not trivial to find the correct parameter that can achieve the intended system functionalities and performance goals [Hubaux et al. 2012]. Moreover, many configuration parameters have complex constraints (e.g., correlations and dependencies with other parameters and the system environments [Benson et al. 2009; Zhang et al. 2014]). If any of the constraints are not followed, the user's setting would become erroneous. Furthermore, the complexity of the systems under configuration adds another level of error proneness. Without carefully examining all the related aspects, it is hard to make configuration settings correct.

[5]Some configuration errors may trigger defects (e.g., software bugs).
[6]For open-source systems, software bugs can also be fixed by the users. However, most of the users do not have time or the ability to understand the code and fix the bugs.

—*Invisibility:* Since users (e.g., system administrators) usually do not or cannot know the system internals, they may not be aware of the inexplicit configuration constraints that are neither documented in user manuals nor informed via system reactions. Xu et al. [2013] report that all the seven studied software projects have inexplicit or even hidden configuration constraints. Moreover, the invisibility of system implementation impairs users' ability to understand the configurations and determine the settings. For example, Rabkin and Katz [2013] report that one-third of the configuration errors in Hadoop systems are memory misconfigurations. One major reason is that users do not have a deep understanding of how Hadoop manages the memory.

—*Dynamics:* Configuration is not a one-time effort but needs to be monitored and tuned from time to time. A perfect configuration may become problematic due to the change of system environments or workloads. Also, software evolution may change configuration semantics, and turn correct settings into erroneous ones [Zhang and Ernst 2014]. In Yin et al. [2011], 16.7%–32.4% of the studied configuration errors occurred in systems that "used to work." The root causes include hardware changes, environment changes, resource exhaustion, and software upgrade.

—*Bad Design and Implementation:* Configuration is one type of system interface exposed to users. Unfortunately, the current practices of configuration design and implementation clearly do not keep this in mind. First, little guidance is provided by the system to help users perform configuration tasks. Among the Linux and eCos users surveyed in Hubaux et al. [2012], more than 56% complained about the lack of guidance on making configuration decisions. Second, as discussed in Section 6, many of the constraints and requirements of configurations are not user friendly but prone to errors. Last but not the least, many of the current systems do not anticipate possible misconfigurations and do not react gracefully to configuration errors (e.g., denying the errors and printing explicit log messages to notify users), as demonstrated in Xu et al. [2013].

3. WHY ARE CONFIGURATION ERRORS DIFFICULT TO DEAL WITH?

The difficulties of tackling configuration errors are rooted in both their inherent characteristics and the lack of tool support for troubleshooting and tolerance.

3.1. Configurations are Complex, So Are the Systems

The complexity of configurations not only induces users' misconfigurations, but also presents significant challenges to approaches that aim at dealing with configuration errors, including prevention, detection, and troubleshooting. The high level of complexity prevents users from carefully understanding and examining each configuration setting one by one. Also, automated tools for misconfiguration detection and troubleshooting have to explore an enormous error space consisting of large numbers of configuration parameters, as well as their correlations and dependencies.

Complexity by Design. Configurations are complex by design, because they consist of a large number of configuration parameters (known as the *complexity of interaction* [Perrow 1984; Reason 1990]). The parameters are often of diverse types and semantics, ranging from feature selection to resource provisioning to performance tuning, as listed in Section 2.1. For example, MySQL database server (version 5.6) has 461 configuration parameters controlling different features, buffer sizes, time-outs, resource limits, etc.; similarly, Apache HTTP server (version 2.4) has more than 550 parameters across all the modules. Moreover, as shown in Xu et al. [2015], the number of configuration parameters of today's software keeps increasing at a rapid rate with software evolution. Take Hadoop as an example; the number of the parameters of MapReduce has grown more than nine times (from 17 to 173) from Apr. 2006 to Oct. 2013. They also observe that the configuration parameters are added with new software versions released, but are removed at a much slower rate, probably due to backward compatibility concerns, or developers' lack of sufficient knowledge or confidence in removing parameters introduced by someone else [Xu et al. 2015].

Similarly, many parameters have a complex value space that is hard for users to set correctly. For example, the value space of an integer number (such as buffer sizes) could be [0, `INT_MAX`]. It is not obvious for users to know the effect of different settings.

The complexity of configurations is also reflected by the correlations and dependencies among different configuration settings (known as the *tightness of coupling* [Perrow 1984; Reason 1990]). Zhang et al. [2014] report that 27%–46% of the configuration parameters in their study have correlations with the other parameters. Benson et al. [2011] report that the complexity from the dependencies of network-device configurations grows steadily over time, with a factor of 2 over a 12-month timespan in the worst case.

Such complexity that results in the explosion of error space presents significant challenges to users' understanding and checking configuration settings, to developers' testing potential problems caused by configuration errors, and to support engineers' misconfiguration troubleshooting.

Complexity of systems and environments. The complexity of configurations is also derived from the complexity of the systems under configuration, as well as their execution environments. First, the complexity of systems and environments makes it hard to quantify the effect of configuration settings. Even with the same configuration settings, different workloads or environments (e.g., resource utilization) would lead to different results or performance. This poses challenges for checking the correctness of users' configuration settings, as discussed in Section 3.2.

Moreover, the configuration settings of one system may have correlations or dependencies with its corunning systems and the underlying system stacks. Welsh [2013] describes a crash case in Google where a system's configuration change (leading to larger numbers of socket connections) causes failures of another system due to the exhaustion of file descriptors. In the study of Yin et al. [2011], 21.7%–57.3% of the studied configuration errors involve configurations beyond the systems or spanning over multiple hosts. Such cross-system configuration errors are especially challenging to deal with because they are mostly hidden and unexpected.

3.2. It Is Hard to Ensure Correctness

Due to the complexity of configurations, it is challenging to ensure the correctness of configuration settings before deploying them to the production. The enormous error space prohibits manually checking all possible errors. Also, traditional approaches to checking software correctness, including bug detection and software verification, are not applicable, because configuration errors are not defects inside the software—even perfect, bug-free software can be misconfigured.

Many of today's software projects only provide basic sanity checks to detect syntax-related configuration errors. Surprisingly, many *illegal* configuration errors that violate predefined constraints can escape from the checking procedures and cause system failures [Yin et al. 2011; Xu et al. 2013]. In Yin et al. [2011], 38.1%–53.7% of the studied configuration errors are illegal and can be potentially detected by rule-based checkers. The challenges of detecting illegal configuration errors lie in the collection and maintenance of the correctness rules and constraints. Obviously, manually writing rules and constraints for every configuration parameter is tedious, error prone, hard to be complete, and hard to keep updated with code changes.

On the other hand, many configuration errors (e.g., 46.3%–61.9% in Yin et al. [2011]) have *legal* values in terms of syntax and semantics. They are incorrect because they do not match workloads or runtime environments, or do not deliver the functionalities/ performance desired by users. For example, in Hadoop, the default setting of the maximum JVM heap size (200 megabytes) is perfectly legal; however, it may not be sufficient for heavy Hadoop jobs and may cause `OutOfMemoryError`. Therefore, detecting legal configuration errors is challenging and requires more information.

Testing is another common practice of checking the correctness of configuration settings. The configuration settings are usually first tested in a testing environment, and then deployed to the production systems (if they pass the testing oracles). However, it is prohibitively difficult for the testing environment to simulate the production systems because of the different scale, workloads, and resource constraints. As pointed out by Welsh [2013], *"hard' problems always come up when running in a real production environment, with real traffic and real resource constraints."*

3.3. Log Messages Are Often Cryptic or Absent

Explicit, pinpointing log messages are particularly helpful to deal with configuration errors. If the system could pinpoint the configuration error with explicit log messages, users could directly reset the configurations and fix the error without resorting to the technical support. Even if the user misses the log message and reports the problem, the diagnosis would be efficient. Yin et al. [2011] study the diagnosis time of configuration-related issues and find that log messages that pinpoint configuration errors can shorten the diagnosis time by 3–13 times as compared to the cases with ambiguous messages, or by 1.2–14.5 times as compared to cases with no messages.

Unfortunately, the log messages are often cryptic, absent or even misleading, which increases the difficulties of resolving configuration errors. Yin et al. [2011] report that only 7.2%–15.5% of the studied configuration issues have explicit log messages that pinpoint the configuration error.

First, configuration errors are often manifested through "cryptic" log messages, that is, logs only containing failure information (e.g., stack traces) without pinpointing the misconfigured parameter or the erroneous settings. Such log messages are difficult for users or even support engineers to reason out the root causes. Rabkin and Katz [2011a] show that the default configuration setting of `fs.default.name` in Hadoop (version 0.20.2) will crash the system with a `NullPointerException`. Although the error logs contain the stack trace recording the failure executions, it is not easy for users to examine the traces and reason out the error. Xu et al. [2013] give a real-world case where the ambiguous symptoms misled the support engineers to mistake a configuration error for a software bug, which resulted in unnecessarily high support cost. Jiang et al. [2009] observe that misconfigurations tend to have too many noisy, unrelated log events in the studied storage systems, making it less effective for problem root cause prediction based on log events.

Moreover, Zhang and Ernst [2013] show that configuration errors can lead to silent, no-crashing failures—ones that do not exhibit a crashing point, dump a stack trace, output an error message, or indicate suspicious program variables. The lack of such information makes many existing debugging techniques such as dynamic slicing, dynamic information flow tracking, and failure trace analysis inapplicable, because these techniques need the crashing points as inputs.

The log messages sometimes are even misleading. Yuan et al. [2011b] shows a real-world example from Apache that a configuration error resulted in a misleading log message showing "no space left on disk," while the user's file system and disk were perfectly healthy with plenty of free space. The root cause was the misconfiguration of the mutex mechanism.

3.4. Users Lack Troubleshooting Support

When configuration errors are manifested through failures, users (e.g., system administrators) have to troubleshoot the errors in order to correct them. Troubleshooting configuration errors involves analyzing the failure site (e.g., log messages, core dumps, stack traces) to reason out the erroneous configuration settings (the root cause). Due to the enormous complexity and error space, troubleshooting configuration errors is often a time-consuming and frustrating "needle-in-a-haystack" process. The situation is worse for some configuration problems, such as performance anomalies and silent failures, which do not have explicit error manifestation and require dynamic profiling for diagnosis.

Unfortunately, there is limited tool support that can help users troubleshoot configuration errors. This is very different from software debugging, which has a rich set of available tools. Despite the similar process that tries to link failures to the root causes, troubleshooting configuration errors poses unique challenges in the following aspects.

(1) *Users cannot debug in the same way as developers.* First, unlike developers, the debugging information (e.g., source code) is not always available to the users like system administrators (e.g., in commercial software and complied binaries). Second, the users may not have the expertise in debugging complex systems (cf. Section 2.2). Even if the system is open sourced, most of the users still cannot debug or do not have time to debug. For example, configuration errors dominate the number of customer issues in the open-sourced Hadoop systems [Rabkin and Katz 2013].

(2) *Recognizing errors inside the software is not enough.* More fundamentally, the goal of debugging is to recognize the errors (e.g., null-pointer dereference, double-free errors) inside the software. However, configuration errors are outside the software itself. There is still a big gap between the error inside the software and the errors inside configuration settings. Filling the gap is not trivial for users who have limited understanding of system implementation.

Due to the lack of tool support, current misconfiguration troubleshooting still mainly relies on manual examination based on support engineers' experiences. Some users apply the trial-and-error method, which is inefficient and may have side effects harming system dependability [Su et al. 2007]. In many circumstances, to resolve configuration-related failures, rolling back to a previous good state is the current band-aid fix in large systems, mainly because there is no better way.

3.5. Traditional Fault-Tolerance and Recovery Techniques Can Hardly Help

Configuration errors are harder to tolerate, compared with hardware failures and software bugs. Many of them impair the system as a whole rather than one or two computing nodes or data replicas. Thus, traditional fault tolerance techniques such as data/modular redundancy and Byzantine fault tolerance can hardly mask configuration errors [Attariyan and Flinn 2010]. For example, Oppenheimer et al. [2003] observe that data redundancy is less effective in masking failures caused by configuration errors, because they tend to be performed on files that affect the operation of entire Internet services. In Song et al. [2009], five out of nine ZooKeeper outages in Yahoo! were caused by configuration errors, while none of them would have been tolerated if a Byzantine-fault-tolerance protocol were used. Tolerating configuration errors often requires redundancy on the system's control panel. For example, Mahajan et al. [2002] report that providing multiple configuration choices is effective in hiding the impact caused by BGP misconfigurations.

It is also hard to recover from failures caused by configuration errors. For example, rebooting, which reclaims system resources and clearing runtime states, is less

Table I. The Overview of Existing Approaches to Tackling Configuration Errors

(a) Approaches for system development		
Stage	Category	Index
System design and implementation	Building configuration-free systems	Section 5
	Making systems easy to configure	Section 6
Quality assurance	Hardening systems against configuration errors	Section 7
(b) Approaches for system management		
Scenario	Category	Index
Setting and validation	Checking the correctness of configurations	Section 8
Deployment and monitoring	Automating deployment and monitoring procedures	Section 9
Coping with failures	Troubleshooting configuration errors	Section 10

helpful because configuration errors usually change the persistent system state. Rolling the system back to a recorded healthy state (snapshot) could be a potential solution based on techniques such as application checkpointing, backup/restore, and systemwide undo [Brown and Patterson 2003]; however, the runtime overhead and storage requirement may not be negligible.

The limitations of both the tolerance and recovery techniques underscore the importance of quickly troubleshooting and fixing configuration errors. After all, tolerance and recovery techniques can only reduce the impact of configuration errors rather than pinpointing or fixing them.

4. AN OVERVIEW OF EXISTING APPROACHES

In this article, we discuss the state-of-the-art systems approaches to tackling configuration errors across a broad spectrum of system development and management processes, including system design and implementation, quality assurance, correctness checking, deployment, and troubleshooting. Table I gives an overview of the approach categories in the context of each stage/scenario. Our goal is to help the configuration stakeholder (including both system vendors and system users) understand the challenges lying in each stage/scenario and the potential solutions they can leverage.

4.1. Approaches for System Vendors in the Development Stages

System development refers to building and maintenance of the system software. A development cycle includes design, implementation, documentation, testing, etc. Configuration, as a part of the system, certainly follows such standard development processes. Since configuration errors cause high financial and human costs (cf. Section 1), a number of efforts have been made to help system vendors tackle configuration errors during system development. In comparison to the approaches for system management, addressing configuration errors at the development stages can *fix* the errors in the first place with low cost of change. These efforts mainly include the following three aspects:

—*Building configuration-free systems:* A natural idea of solving configuration errors is to remove the need of configuration by autoconfiguration. This prevents users from misconfiguring the systems in the first place. However, not all the configuration settings can be automated; also, it may be difficult for autoconfiguration solutions to balance simplicity and flexibility, the two conflicting goals of system design. We discuss the approaches toward configuration-free systems in Section 5.

—*Making systems easy to configure:* A more tangible goal is to make systems easy to configure in order to lower the error rate. This requires designing and implementing configurations from the user's perspective; for example, designing user-friendly configuration languages, and making configuration requirements intuitive and easy to follow. We discuss the approaches that aim to make user-friendly configurations in Section 6.

—*Harden systems against configuration errors:* Despite the aforementioned efforts, we have to confront the fact that configuration errors, as human errors, are inevitable [Patterson et al. 2002]. A practical way of tackling these inevitable errors is to harden the systems against such errors. Ideally, the system should deny users' misconfigurations and pinpoint the errors to users. We discuss the approaches to hardening systems, mainly at the stage of quality assurance, in Section 7.

4.2. Approaches for System Users in Different Usage Scenarios

The typical usage scenarios regarding configurations include setting and validating configuration values, deploying and monitoring configuration settings to the production systems (often in a larger scale), and diagnosing system failures caused by configuration errors (if any). Many of the activities related to configuration are tedious, time consuming, and prone to errors if performed manually. Thus, a number of supporting tools have been developed to automate the processes.

—*Checking the correctness of configuration settings:* As discussed in Section 3.2, it is prohibitively difficult to ensure the correctness of configuration settings. Fortunately, current approaches are able to provide automatic solutions to *detect* or *expose* certain types of configuration errors in users' settings, before deployment and roll-out. These approaches include static rule-based misconfiguration detection and dynamic testing-based validation. We discuss these approaches in Section 8.

—*Automating the deployment and monitoring:* Many of today's systems in production, exemplified by Hadoop clusters and data centers, are highly distributed and large in scale, and consist of hundreds, even thousands, of computing nodes. It is tedious and extremely error prone for administrators to manually deploy configuration settings to every node and to keep track of them. A number of tools have been built to automate the processes, which is discussed in Section 9.

—*Troubleshooting configuration errors:* To fill the vacancy of troubleshooting support (cf. Section 3.4), a number of research efforts have been made to provide solutions and tool support to help administrators identify the configuration errors that result in system failures. We discuss the approaches to troubleshooting configuration errors in Section 10.

5. BUILDING CONFIGURATION-FREE SYSTEMS

One natural idea of resolving configuration errors is to free system administrators from configuration by automating configuration processes. This lifts the users' burden of configuration and prevents human errors in the first place. However, building configuration-free systems is not trivial. First, not all the configurations can be or should be automated. For example, it is hard for systems to automatically determine the configurations that require information outside the system, e.g., connection information of backup services; also, systems should not automatically decide critical configuration settings that may cause permanent changes in the system state or user data [Verbowski et al. 2006b].

Moreover, it is difficult for configuration automation to balance *simplicity* of configuration and *flexibility* of control, the two conflicting goals of system design. On one hand, system users desire configurations to be simple and easy to use; on the other hand, many of them need the knobs to customize and fine-tune the systems, especially in performance-critical systems.

Despite the preceding challenges, many efforts have been made toward making system administrators "free of configuration," including eliminating configuration parameters, automatically selecting (and recommending) configuration values, and recording and replaying others' configurations.

5.1. Eliminating Configurations

Configurations are exposed to users for providing the flexibility of control. Since the flexibility comes with a high cost (i.e., complexity and error proneness), the following two types of configurations should be eliminated: (1) configurations that are not needed by most of the users; and (2) configurations that are hard for users to set correctly. After all, if users do not or cannot leverage the flexibility provided by the configurations, the cost is not worthwhile.

Eliminating unused configurations. Configuration parameters that are seldom set by users should be hidden or be removed, because they make configurations more complex without producing much benefit in terms of user-desired flexibility. Lots of flexibility can lead to lots of confusion.

Xu et al. [2015] report that 31.1%–54.1% of the configuration parameters were seldom set by any users in four mature, widely used system-software projects. Hiding or removing these parameters can significantly simplify configurations with little impact on existing users. Note that the complexity caused by "too many knobs" does come with a cost—up to 48.5% of user-reported configuration issues are about users' difficulties in finding or setting the parameters (from the entire parameter set) to achieve their desired functionalities or performance goals; up to 53.3% of configuration errors are introduced due to users' staying with default values incorrectly. Based on their results, they propose a set of design principles that can significantly reduce the configuration space.

In fact, some parameters could be automatically set according to the information of the current system instance. For example, PostgreSQL has a number of *preset* parameters whose values are determined during the installation of the software. Take `block_size` (the PostgreSQL data-block size) as an example: its value is automatically set to be aligned with the size of the file system's disk blocks in order to optimize the disk usage.

To understand which configurations the users need, system vendors should maintain user-feedback loops for configuration design. Right now, many system vendors are collecting users' configuration settings for troubleshooting purposes [Wang et al. 2003; Jiang et al. 2009]. Such data provide great opportunities to understand users' configuration settings in the field, including how the users set each parameter and which parameters are rarely set.

Eliminating hard-to-set configurations. System developers also attempt to eliminate configurations that are hard for users to set correctly by new system design or configuration automation. The memory management of Apache Flink (a large-scale data processing engine, previously known as Stratosphere [Alexandrov et al. 2014]) is one great example of eliminating error-prone configurations with better design. Flink effectively prevents `OutOfMemoryError` using its novel serialization-based JVM memory management, which "*makes the difference between a system that is hard to configure with unpredictable reliability and performance and a system that behaves robustly with few configuration knobs*" [Hueske 2015]. As reported by Cloudera Inc. [Rabkin and Katz 2013], memory misconfigurations (causing `OutOfMemoryError`) contribute to approximately a third of all configuration problems of Hadoop, as manually setting JVM heap memory is error prone and fragile, especially if the workload characteristics or the execution environment change.

In addition, system developers attempt to automate configuration settings to eliminate ones with tedious setting procedures and with difficult setting policies (e.g., trade-offs between system and environmental conditions). Hippodrome [Anderson et al. 2002] automates the iterative configuration process of enterprise-scale storage systems to cope with the diversity of workloads (the configurations mainly include the provisions of hard drives and logical units). Hippodrome analyzes a running workload to determine its requirements, calculates a new storage system configuration, and then migrates the existing system configuration to the new configuration. Hippodrome makes better configuration decisions by systematically exploring the large space of possible configurations based on the predefined storage system models. MUSE [Chase et al. 2001] applies similar principles to generate the service and server configurations toward provisioning energy-conscious resources for Internet hosting data centers. DAC [Chen et al. 2010] automates IP-address configurations for datacenter networks, considering both the locality and topology information for performance and routing purposes. In this way, it eliminates the error proneness of traditional address configuration protocols such as Dynamic Host Configuration Protocol (DHCP) that require a huge amount of manual inputs.

5.2. Automatic Performance Tuning

One specific class of hard-to-set configuration parameters are those related to system performance. Large systems usually include a large number of performance-related parameters (e.g., memory allocation, I/O optimization, parallelism levels, logging). Their impact on performance is often poorly understood due to the inexplicit correlations and interactions both inside the system and across multiple system components and runtime environments. Consequently, tuning system performance is challenging, especially for users with limited knowledge of system internals (cf. Section 2.4). Also, with the explosive configuration space, it is prohibitively difficult and costly to test out every value combination and then select the best ones.

Mature systems provide *default* values for these configuration parameters. The default values are usually carefully selected by developers based on their experience and/or in-house performance testing results. Ideally, good default values can satisfy users with common workloads and system/hardware settings. However, given a particular workload and system runtime, it is hard for the static default values to deliver the *optimized* system performance.

There is a wealth of literature on performance tuning by selecting configuration values. The basic idea is to model the performance as a function of configuration settings, as in the following equation:[7]

$$p = F_W(\vec{v}),\ \text{where } \vec{v} = (v_1, v_2, \ldots, v_n). \tag{1}$$

Here, F_W is the performance function of a workload type W; $\vec{v}$ is the configuration value set consisting of the value of each i-th configuration parameter (the number of parameters is denoted as n); and p is the performance metric of interest (e.g., execution time, response latency, throughput). With such a model, the performance tuning problem is to find the value set $\vec{v}^*$ that achieves the best performance (i.e., *argmax*) in terms of the metric, that is,

$$\vec{v}^* = argmax\ F_W(\vec{v}). \tag{2}$$

Since the performance function F_W is often unknown or does not have a closed form, a number of *black-box* optimization algorithms have been proposed, including the recursive random algorithm [Ye and Kalyanaraman 2003], the hill-climbing algorithm [Xi

[7]This model can be extended to include other factors. For example, Herodotou and Babu [2011] models the performance of one MapReduce job W as $p = F_W(\vec{v}, \vec{r}, \vec{d})$ where $\vec{r}$ is the allocated resources and $\vec{d}$ represents other statistical properties of data processed by W.

et al. 2004], the neighborhood-selection algorithm [Osogami and Itoko 2006], and gridding [Herodotou and Babu 2011]. Other studies try to capture the performance characteristics using specific models. For example, Duan et al. [2009] use a Gaussian process to represent the response surface of F_W in relational database systems; Zheng et al. [2007] apply dependency graphs to describe the performance dependencies among different parameters of Web applications.

These studies provide theoretical guidance for modeling performance impact of configuration settings. However, it is fundamentally difficult to precisely model the system performance in the field due to many unexpected and confounding factors. Consequently, some of the models are limited to certain workloads and environments, making them less appealing in practice.

5.3. Reusing Configuration Solutions

As configuration is difficult, the experience and efforts toward the correct configuration solutions should be shared and reused. Often, a configuration task difficult for some users has been encountered and solved by other users. Most importantly, the previous working solutions are likely to be helpful (and even directly applicable) to the new systems under configuration.

Configuration File Templates. The basic form of reusing configurations is through *templates*. A typical configuration file template is a working solution for one particular use case. For example, Squid Web proxy provides a number of configuration file templates for different usage scenarios including reverse proxy, filtering for instant messaging (chat programs), filtering for multimedia and data streaming, etc. Squid users can start with the templates based on the upper-level applications, and edit system-specific parameters accordingly. This saves a lot of effort, in comparison to configuring the system from scratch. Templates can also be made for different hardware or environment conditions. For example, MySQL once provided five templates for servers with different memory sizes, ranging from less than 64MB to larger than 4GB. These templates save users' efforts of tuning memory-related configuration parameters, which is difficult but critical to performance.

In fact, templates not only can be provided by system vendors, but also can be distributed by third-party tool providers or among user communities. For example, the user communities of configuration management tools (e.g., Puppet, Chef, and CFEngine) discussed in Section 9 have the tradition of sharing configurations for a variety of common software systems.

Record-and-Replay Systems. Static configuration file templates have two major limitations. First, they cannot capture configuration *actions*, for example, setting file permissions, installing software packages, creating new user groups. These actions can be a necessary part of the configuration solution. Second, the solution provided by the templates may not work for the new system; applying a wrong solution may have side effects—changes of the system states. In this case, the changes need to be undone. Manually undoing system changes is not only tedious but also difficult as users may not be aware of some implicit changes of system states.

Record-and-replay systems are proposed to address the preceding two limitations and to make configuration reuse fully automated. The basic idea is to *record* the traces of a working configuration solution on one system, and then *replay* the traces on other systems under configuration. If the replay fails (e.g., not passing predefined test cases), the system will be automatically rolled back to its original state. Since the traces for the same configuration problem may differ (e.g., due to system and environment settings), multiple raw traces could be merged to construct the canonical solution. A centralized

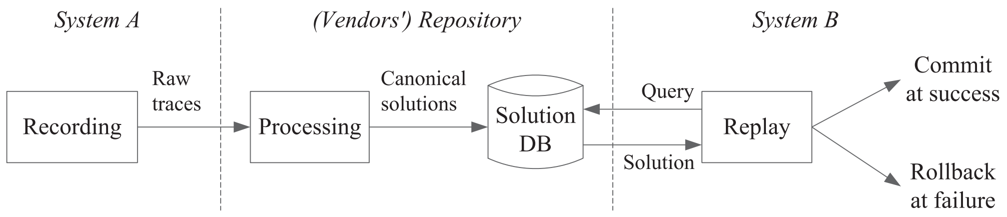


Fig. 1. Main components and workflow of a record-and-replay system.

database is usually deployed to maintain all the solutions for easy retrieval. Figure 1 illustrates the main components and the workflow of record-and-replay systems.

AutoBash [Su et al. 2007] and KarDo [Kushman and Katabi 2010] are two representative record-and-replay systems designed for different use cases. AutoBash is designed for recording and replaying configuration tasks performed in Bash (or other UNIX-style shells). It leverages kernel speculation to automatically try out solutions from the solution database one by one until it finds a working one for a configuration problem. The kernel speculator ensures that the speculative state is never externalized, that is, it isolates AutoBash's replay activities from other nonconfiguration tasks. AutoBash requires users to provide test cases and oracles (called predicates) in order to decide whether or not the configured system is correct. Later, Su and Flinn [2009] further propose automatic generation of predicates by observing the actions of users' troubleshooting processes.

KarDo is designed for automating Graphical User Interface (GUI) -based configurations on personal computers. It records the window events when users perform configuration tasks based on OS-level accessibility support. KarDo analyzes the raw traces and identifies the window events specific to the task and the events for state transition events. It then constructs the canonical solution for a configuration task; the canonical solution works for systems with different internal states.

6. MAKING SYSTEMS EASY TO CONFIGURE

As it is challenging to create systems completely free of configuration, a more practical direction is to make systems easy to configure in order to lower the human-error rate. The key toward this goal is to treat configuration as user interfaces and adopt the user-centric philosophy for configuration design and implementation. In this section, we discuss two classes of efforts toward making configurations easy: declarative configuration design and user-friendly configuration constraints. The former enables users to use high-level configuration primitives rather than minutiae of settings, while the latter makes the configuration process intuitive and less prone to errors.

Please note that making system easy to configure is not, and should not be, limited to these two aspects. With the user-centric design philosophy, the tool building and support should be based on the understanding of users' difficulties and pains regarding their configurations. This may require user interviews [Velasquez et al. 2008] and field studies [Barrett et al. 2004].

6.1. Declarative Configuration

The idea of declarative configuration design is to help users state *what* is to be configured, but not necessarily *how* it is to be configured. In this way, users can work with high-level primitives instead of the minutiae of configuration settings. The challenges lie in designing the right level of abstraction. What is needed, on one hand, is to empower users to describe configuration goals without getting bogged down in how to achieve those goals. On the other hand, declarative approaches need to automatically translate the declared goals into low-level configuration actions.

DeTreville [2005] proposes to make system configuration declarative. This is done with a hierarchical system model to express rules that define how various components (submodels) compose a system. The models are in the form of functions, defining the configuration constraints between different components. The system parameter settings, as the inputs of these model functions, produce a system instance. With this model, users can establish system policies to define whether a system instance is acceptable or not, e.g., "a system model is valid only when all its submodels are valid." Following that, Dolstra and Hemel [2007] propose a pure functional model to express the configurations of software packages and their dependencies.

These ideas have been integrated in a number of configuration management tools such as Puppet, Chef, and CFEngine (cf. Section 9). For example, the following Puppet configuration snippet declares two system services: `apache` and `apache-ssl`. The former requires a package installation, while the latter is based on the former service and requires an additional file. Puppet would ensure the dependencies according to the declarations.

```
class apache {
    service{'apache': require => package['httpd'] }
}

class apache-ssl inherits apache {
    # Host certificate is required for SSL to function
    service['apache'] { require +> file['apache.pem'] }
}
```

Declarative methods are also applied to resource configuration. Hewson et al. [2012] design an object-oriented configuration language that allows administrators to provision physical machines by describing the constraints (e.g., hardware limit) and requirements of resources. It translates users' declarations into a Constraints Satisfaction Problem (CSP), and uses a CSP solver to find a valid solution. Other tools model the declared problems into Boolean Satisfiability Problem (SAT) [Narain 2005, 2008] and Constraint Logic Programming (CSP) [Goldsack et al. 2009].

Most existing declarative configuration approaches only allow users to declare the procedures instead of expressing their *intentions* of configuration. As the next step, we envision a breakthrough in the direction of helping users express high-level intentions. For example, users can express an intention such as *"I want my data to be more reliable"*; accordingly, the system can suggest, or automatically adjust, relevant configuration parameters such as RAID level or number of data replicas.

6.2. Design and Implementation of User-Friendly Configuration Constraints

Configuration constraints are correctness rules that differentiate correct configurations from misconfigurations. In order to conduct correct configurations, users' settings have to strictly follow the constraints defined by developers. Thus, it is critical for developers to design and implement user-friendly constraints for configurations, which can reduce users' confusion and mistakes.

Design principles. The constraints are one part of the configuration interface presented to the users, and their design should follow the interface design principles (e.g., Norman [1983a, 1983b], Molich and Nielsen [1990], and Nielsen and Molich [1990]). In particular, the following principles should be carefully applied and evaluated in the design of configuration constraints,

Software (version):
Hadoop MapReduce (2.2.0--2.3.0)

Configuration parameter:
`mapreduce.framework.name`

Error-prone Constraint:
The settings of this parameter have to be lower-case letters, otherwise the system cannot recognize the framework.

Real-world misconfiguration:
Case #19571962 on ServerFault.com: The user got runtime exception because of the following misconfiguration:

```
<name>mapreduce.framework.name</name>
<value>Yarn</value>
```

(a) Non-intuitive

Software (version)
Apache HTTP Server (2.0--2.4)

Configuration parameter:
`MaxMemFree`

Inconsistent Constraint:
The unit of this parameter setting is *kilobytes*, while most of the other memory-related parameters of Apache, such as `RLimitMem`, `ThreadStackSize`, `SendBufferSize`, are with the unit *bytes*.

Real-world Problems:
A user was confused by the size of his setting, and another user was confused by the program behavior (i.e., multiplying user settings by 1024) when comparing with `ThreadStackSize`.

(b) Inconsistent

Fig. 2. Real-world examples of configuration constraints that are not user friendly.

—*Intuitiveness:* Being intuitive means to *speak the users' language* and to *minimize the users' memory load* [Molich and Nielsen 1990]. With the large number of configuration parameters, it is not reasonable to assume that users remember every detailed constraint. In the example in Figure 2(a), the developers expect too much of users who do not read the parsing code.

—*Consistency:* The constraints of different configuration parameters should be consistent. Since computer system users tend to derive operations by analogy with other similar aspects of the system [Norman 1983b], inconsistent constraints of similar types of configurations would lead to users' confusion and misconfigurations, as shown in Figure 2(b).

—*Providing feedback:* When users' configuration settings violate constraints, the system should pinpoint the error and provide potential solutions via the display (e.g., `stdout`, `stderr`) or log messages. As reported by Yin et al. [2011], good log messages can shorten users' self-diagnosis time by up to an order of magnitude.

—*Minimalism:* The constraints should be designed as simple as possible as long as they can satisfy users' needs. For example, developers often choose overflexible data types for configuration parameters (e.g., using numeric types instead of the simple Boolean or enumerative types). However, users might not need such flexibility. Often, flexibility comes with the cost of complexity.[8]

—*Help and documentation:* The constraints of each parameter should be rigorously documented not only in user manuals but also in systems themselves. The documentation should allow users to search the constraint information in a convenient way.

Unfortunately, many of today's mature software projects do not satisfy these basic design principles. In Xu et al. [2013], all of the six studied software systems (including a commercial system and six open-source server applications) have different levels of inconsistency, silently ignoring or changing users' settings, and undocumented constraints.

[8]Elphinstone and Heiser [2013] share an example as one of the lessons learned in 20 years of L4 microkernel development: the IPC time-out was changed from a floating parameter (supporting the values of zero, infinity, and finite values ranging from 1ms to weeks) into a Boolean one (zero or infinity). It is because the time-outs added complexity (for managing wake-up lists) but were of little use—*"There is no theory, or even good heuristics, for choosing timeout values in a non-trivial system, and in reality, only the values zero and infinity were used."*

Implementation notes. Implementation should also be taken care of to embody the design principles. Developers should always anticipate users' configuration mistakes, which means disciplined checking and logging. The following code snippets come from the configuration parsing procedure for Boolean parameters in Squid Web proxy (version 2.3.5). With such a parsing procedure, Squid silently overrules any users' setting (converting it to "off") as long as the setting is not equal to "on," even for settings with the completely different semantic meaning such as "yes" or "enable." Such implementation should be avoided because it can easily confuse users due to the mismatching between the system behavior and their intention.[9] In latter versions (after we reported the problem), Squid carefully checks and and prints log messages if the setting is neither "on" nor "off."

```
/* squid-2.3.5: src/cache_cf.cc */
if (!strcasecmp(token, "on")) {   //"token" stores users' settings
      var = 1;
} else {
      var = 0;
}
```

Xu et al. [2013] point out that some "unsafe" APIs should be avoided in configuration parsing procedures, such as `atoi`, `sscanf`, and `sprintf`. These APIs do not provide error checking and would translate users' erroneous settings to undefined values. Again, developers cannot assume users' settings are always correct [Zeller 2009]. Instead, a good practice is to use safe APIs such as `strtol` and to check errors through `errno` and end pointers (`endptr`).

7. HARDENING SYSTEMS AGAINST CONFIGURATION ERRORS

Despite the aforementioned efforts, the cruel fact is that configuration errors, as human errors, are inevitable [Patterson et al. 2002]. To cope with such inevitable errors, the systems themselves should anticipate potential configuration errors and be hardened against these errors. Ideally, when the users misconfigure the system, the system should not fail or crash; instead, the system should deny the erroneous settings and print log messages to pinpoint the errors.

To harden systems against configuration errors, researchers propose testing system resilience and reactions to configuration errors as a part of quality assurance for software systems. We refer to such testing efforts as *configuration testing* in this article.[10] Similar to other testing practices (e.g., unit testing, functional testing) that try to expose software defects, the goal of configuration testing is to expose the system's vulnerabilities to configuration errors (i.e., *misconfiguration vulnerabilities* [Xu et al. 2013]) so that developers can fix the vulnerabilities before releasing and shipping the system to the users. Specifically, *misconfiguration vulnerabilities* are unexpected system behavior under certain configuration errors, including crashes, hangs, missing functionalities, producing incorrect results, and mistranslating users' settings. Fixing these vulnerabilities includes enhancing system resilience to the errors (e.g., adding

[9]MySQL Bug #51631 is a supporting example. Before version 5.1, MySQL defined numeric values 1 and 0 as the only valid settings of the Boolean parameter, `general-log`. Although this value constraint was documented, users still got confused when setting the parameter as "on" or "off." This nonintuitive design is fixed in the latter MySQL versions.

[10]Note that the configuration testing discussed in this section is completely different from the testing efforts that attempt to expose software bugs under different combinations of configuration settings (there is a large body of literature of efficient software testing by reducing the configuration combinatorics). This article focuses on tackling configuration errors rather than defects in the software. Thus, we only discuss the testing to expose systems' vulnerabilities to configuration errors.

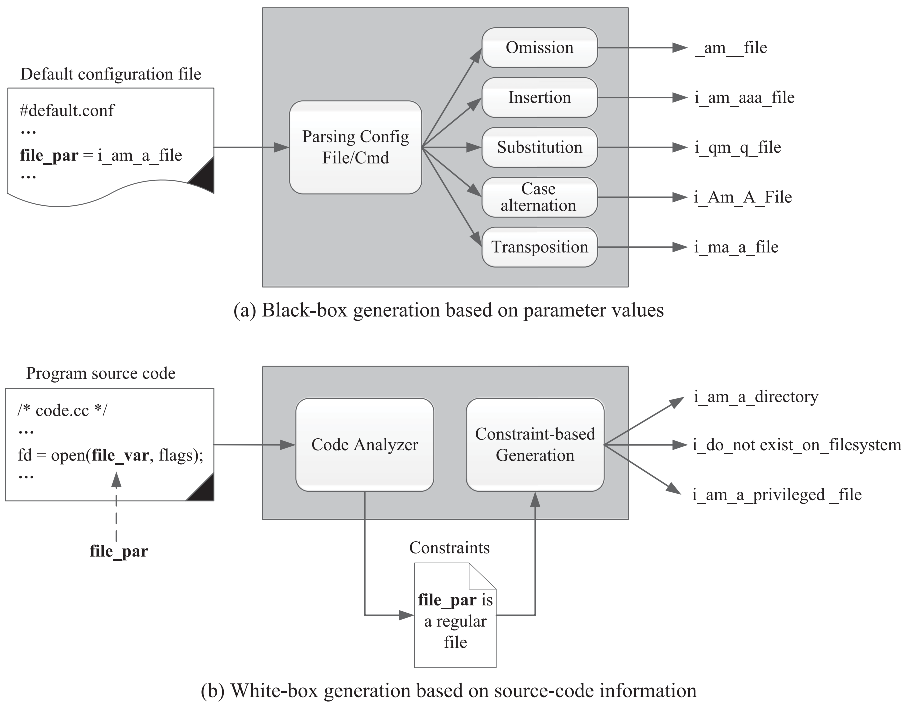


Fig. 3. A demonstrative example of error generation for configuration testing. In this example, `file_par` is a configuration parameter representing a file path; the value of `file_par` is stored in the variable `file_var` in the program, as shown in (b). The generated erroneous settings (i.e., the outputs) will be used to test the system resilience and reactions to these errors.

proactive checking code) and improving system reactions to the errors (e.g., denying the errors and printing error messages).

The main challenge of configuration testing is to anticipate the configuration errors that users will potentially make. With the enormous error space (cf. Section 3.1), it is prohibitively difficult and costly to list all the possible errors, and to test whether or not the system is resilient and could react gracefully to each of them. Thus, the key is to have a small set of representative configuration errors as test cases. The errors used for testing should be *efficient* so that they can expose misconfiguration vulnerabilities in a short amount of time, *realistic* so that they are likely to be made by real-world users, and *systematic* to expose as many vulnerabilities as possible.

To generate efficient, realistic, and systematic configuration errors for testing, both black-box approaches and white-box approaches have been proposed. Figure 3 demonstrates the process of error generation using these two types of approaches.

Black-Box Approaches. ConfErr [Keller et al. 2008] is a black-box configuration testing tool. It uses human error models rooted in psychology and linguistics to generate realistic configuration errors. The configuration errors generated by ConfErr include spelling errors (e.g., omissions, insertions, substitutions, case alterations), structural errors (e.g., reverting two sections in a configuration file), and semantic errors (e.g., violating RFC, inconsistency between functionality). As a "smart" fuzz testing [Miller et al. 1995], ConfErr understands neither the semantics nor the constraints of each configuration parameter. The configuration errors are generated by mutating the correct configuration settings in the default configuration files. Keller et al. [2008] show that

ConfErr is able to expose a number of vulnerabilities in five widely used open-source software systems.

White-Box Approaches. Complementary to black-box approaches, white-box approaches generate configuration errors according to how the configuration parameters are used inside the system programs. Spex [Xu et al. 2013] is a white-box tool that generates configuration errors based on the inference of the constraints of configuration parameters. As discussed in Section 6.2, the constraints define the correctness rules of configuration settings. Spex generates errors by violating the inferred constraints in a variety of aspects. In Figure 3(b), Spex generates erroneous settings including a directory path, a nonexistence path, and a privileged file path for a parameter that is constrained to be a regular file path.

To infer constraints of each configuration parameter, Spex statically analyzes the system program to observe how the parameter is read and used. Once the parameter's usage matches predefined patterns, the corresponding constraint is captured. In Figure 3(b), Spex infers the data type of the `file_par` parameter to be a regular file path by observing its value (stored in the `file_var` variable) falling into the `open()` system call. Compared with black-box approaches, the test cases generated by Spex are more efficient and targeted. Besides static analysis, symbolic execution (e.g., Klee [Cadar et al. 2008]) has great potential to generate a comprehensive set of test cases in terms of code coverage, considering configuration settings as the system inputs.

We would like to point out that the tools discussed previously have their limitations. Their generality makes them not capable of understanding the semantics of either the software or the configuration. For example, though Spex can infer that a configuration parameter is an IP address, it is hard for Spex to know what the correct IP address should be, and thus Spex cannot generate specific types of erroneous IP addresses. We envision domain-specific methods to enhance the capability of these tools to expose more sophisticated misconfiguration vulnerabilities.

8. CHECKING CORRECTNESS OF CONFIGURATIONS

In this section, we discuss the approaches that help check the correctness of configuration settings, including *static* misconfiguration detection and *dynamic* configuration validation.

8.1. Misconfiguration Detection

Misconfiguration detection refers to the process of checking the configuration settings before the errors are manifested through runtime crashes and performance anomalies. Most of today's detection approaches are *rule based*: the detector checks the configuration settings against a set of predefined correctness *rules* (also known as *constraints* or *specifications*). If a configuration setting does not satisfy these rules, it will be flagged as a misconfiguration. The key challenge of rule-based detection is to define and manage effective and useful rules.

Defining rules. Most mature systems implement built-in configuration checking logic to detect syntax and format errors against basic rules. As observed by Nagaraja et al. [2004], the checking is useful in detecting users' configuration mistakes. Some advanced, domain-specific rules have been proposed in different system domains. For example, Xiong et al. [2012] model the data-range rules among multiple parameters. The range model of configuration parameters can help detect and fix invalid settings. Feamster and Balakrishnan [2005] define two general correctness rules forBorder Gateway Protocol (BGP) configurations: the path visibility and the route validity. Based on the two rules, their tool RCC detects BGP misconfigurations by statically analyzing

BGP configurations. ConfValley [Huang et al. 2015] provides a simple, domain-specific language called Configuration Predicate Language (CPL) for cloud system operators to write configuration checking logic in a compact, declarative fashion. Compared with traditional imperative languages, CPL can significantly reduce the efforts of writing checking code (up to 10× reduction of lines of code) in cloud systems.

Generally speaking, manually specified rules face the following two problems. First, it is hard to make predefined rules complete [Kendrick 2012]. In the study of Nagaraja et al. [2004], though Apache's checker detected a number of configuration errors based on predefined rules, it did not check a misconfigured file path, causing `mod_jk` to crash. Second, it is costly to keep the rules updated with the software evolution, since it requires repeating the manual efforts. As demonstrated in Zhang and Ernst [2014], configuration rules could be obsolete with code changes.

Learning rules. To address the problems derived from the manual efforts of defining rules, a number of research proposals provide automatic approaches to learning configuration rules. The basic idea is to learn the "common patterns" from large volumes of configuration settings collected from a large number of healthy system instances. The patterns shared by most of the healthy instances are assumed to be correct, and will be used as the correctness rules for misconfiguration detection. A configuration setting that does not follow these patterns is likely to be misconfigurations.

With this idea, a variety of machine learning techniques are applied to learn configuration rules from different types of configuration data. CODE [Yuan et al. 2011a] learns the access patterns of Windows Registry configurations, which indicate the appropriate configuration events that should follow each context (a series of events). Based on these patterns, CODE can sift through a sequence of configuration events and detect the deviant ones as misconfigurations. Palatin et al. [2006] propose a distributed outlier detection algorithm to detect misconfigured machines (i.e., the outlier machine) in grid systems based on log messages, resource utilizations, and configuration settings. Kiciman and Wang [2004] apply unsupervised clustering algorithms on Windows Registry configurations in order to learn configuration constraints based on the Registry-specific structures and semantics. Bauer et al. [2011] detect misconfigurations of access control policies by applying association rule mining. The association rules identify the subset of resources that appear in multiple access records. Access records that fail to satisfy the learned rules are likely to have configuration errors.

The main concern of automatic learning is the accuracy. Inaccuracy results in false rules that cause false positives in detection. False positives do matter. Bessey et al. [2010] report that in reality, users ignore the tools if the false-positive rate is more than 30%. Please note that misconfiguration detection is performed before system failures and anomalies happen. At the time, the users are usually optimistic and confident. This psychological issue fundamentally makes misconfiguration detection different from the postmortem failure diagnosis (when the users or support engineers are desperate and willing to look at any hints). Unfortunately, learning-based approaches often cannot provide sufficient accuracy and are difficult to tune. Many of the learning-based approaches discussed previously also require manual intervention.

To address the inaccuracy of learning techniques, EnCore [Zhang et al. 2014] adopts a template-based learning approach. In EnCore, the learning process is guided by a set of predefined rule templates that enforce learning to focus on patterns of interests. In this way, EnCore filters out irrelevant information and reduces the false positives. Furthermore, the templates are able to express rules that cannot be learned by state-of-the-art machine learning algorithms.

Despite these advances in misconfiguration detection, we have to confront the fact that configuration errors are still hard be ruled out. Besides the limitations of the existing approaches, some misconfigurations cannot be detected even when the settings are enforced to follow the correctness rules. Such misconfigurations are referred to as *legal* misconfigurations [Yin et al. 2011]. Legal misconfigurations are not rare. Among the 434 real-world parameter-misconfiguration cases studied in Yin et al. [2011], "*a large portion (46.3%–61.9%) of the parameter misconfigurations have perfectly legal parameters but do not deliver the functionality intended by users. These cases are more difficult to detect by automatic checkers and may require more user training or better configuration design.*"

8.2. Online Configuration Validation

Online validation (also known as online testing[11]) helps users observe the effect of their configuration settings by testing them out in a validation environment before rolling out the configuration settings and making them visible to the production systems. This complements static misconfiguration detection (cf. Section 8.1) in testing out the settings in a running system. Thus, it has the potential to catch legal misconfigurations.

Online configuration validation is especially useful when the effect of configuration settings is not well understood or cannot be calculated/estimated. It allows the administrators and operators to apply the trial-and-error methods on different settings. Moreover, performing online validation, as a rehearsal, is necessary for configurations that are mission critical or cannot be rolled back; for example, formatting disks that will clean users' data.

The main challenge of the online validation approaches is to construct a validation environment that has the same (or similar) characteristics of the production systems including the workloads, the execution environments, and the underlying infrastructure. If the validation environment cannot capture these characteristics, the validation results are less trustworthy. However, such construction is very challenging because of resource constraints and the separation from the production environments, especially for large-scale systems. Welsh [2013] shares his experience in Google,

> "*Of course we have extensive testing infrastructure, but the 'hard' problems always come up when running in a real production environment, with real traffic and real resource constraints. Even integration tests and canarying are a joke compared to how complex production-scale systems are.*"

As summarized by Bianchini et al. [2005], the online validation solutions need to be *comprehensive* for determining whether or not the configuration settings are valid, *isolated* to the production environments, and *efficient* in terms of resource usage.

Nagaraja et al. [2004] propose integrating the validation component as an extension of the production systems, which addresses the separation between the validation and the production environments. In their prototype designed for multitier Web services, the entire system is divided into two logical slices: the *online slice* that hosts the production services and the *validation slice* that validates new configuration settings. The two slices are connected through a proxy that duplicates user requests from the production services to the validation components. In this way, the validation components can test new configurations under real workloads. The same idea is applied in Oliveira et al. [2006] to validate configurations for database systems.

Due to the runtime overhead, integrating validation into production may be too costly for systems compete for performance. Also, it may not be applicable to distributed systems at a large scale (such as the cloud and data-center systems). It remains an open

[11]The user-side testing for the purpose of configuration validation is different from the testing for exposing the system's vulnerabilities to configuration errors discussed in Section 7.

problem to design and implement effective validation methods for complex, large-scale systems.

9. CONFIGURATION MANAGEMENT

Many of today's systems in production, exemplified by cloud and data-center systems, are highly distributed and large in scale—they consist of hundreds, even thousands, of computing nodes. It is tedious and extremely error prone for administrators to manually deploy and manage configuration settings for every single node. Automatic tools are built to automate the management processes.

In this section, we give an overview of the effort and tool support for configuration management in terms of configuration deployment and monitoring. As both of the problems are well defined, configuration management is a mature field, with a number of widely used tools available.

9.1. Automatic Deployment

Over the last two decades, the system administration community has proposed and developed a number of tools for automatic configuration deployment including LCFG [Anderson 1994], CFEngine [Burgess 1995], NewFig [LeFebvre and Snyder 2004], BCFG [Desai et al. 2005], SmartFrog [Goldsack et al. 2009], etc. Many of the ideas have been encoded in today's mature, well-developed tools, such as Puppet, Chef, and CFEngine3. With the proliferation of distributed and cloud applications, these tools have obtained significant popularity with active user groups.

Most of these tools share the same high-level idea for configuration deployment. At the *front-end*, the tool provides administrators with a declarative language (discussed in Section 6.1), based on which administrators can configure resources (e.g., machines, software, services) and policies (e.g., the rules of using resources) in a concise way. At the *back-end*, the tool automatically translates users' configurations into settings and actions (e.g., starting services, installing software packages).

The declarative languages often follow an object-oriented design, in order to match the characteristics of clusters and data centers where a number of machines share a part of configurations but are different in the site-, group-, or machine-specific configurations. As demonstrated in Figure 4, system administrators usually take a configuration "recipe" (provided by vendors or peer administrators), and customize their configurations in the level of *sites*, *groups*, and *machines*. The tool automatically compiles the recipe and the customizations to generate per-machine configurations. Lastly, it automatically deploys the configuration settings to every single machine under configuration. Such deployment process is systematic and less prone to errors, because it takes off administrators' burden of configuring every single machine.

Anderson and Smith [2005] look back at the reasons why some of the old configuration management tools have failed to move forward, and propose the standards for interoperation of different tools. Delaet et al. [2010] survey a number of configuration management tools using a comparison framework that defines the usability of these tools. Lueninghoener [2011] provides some basic guidance and practices on how to begin using the configuration management tools.

9.2. Monitoring Configuration Changes

Another important aspect of configuration management is monitoring the configuration changes of the system. The critical configuration changes (e.g., changes that affect the persistent system state) should be notified to users in time and be confirmed by users. LiveOps [Verbowski et al. 2006b] is a system management service from Microsoft. It audits the interactions between applications and the persistent system states using

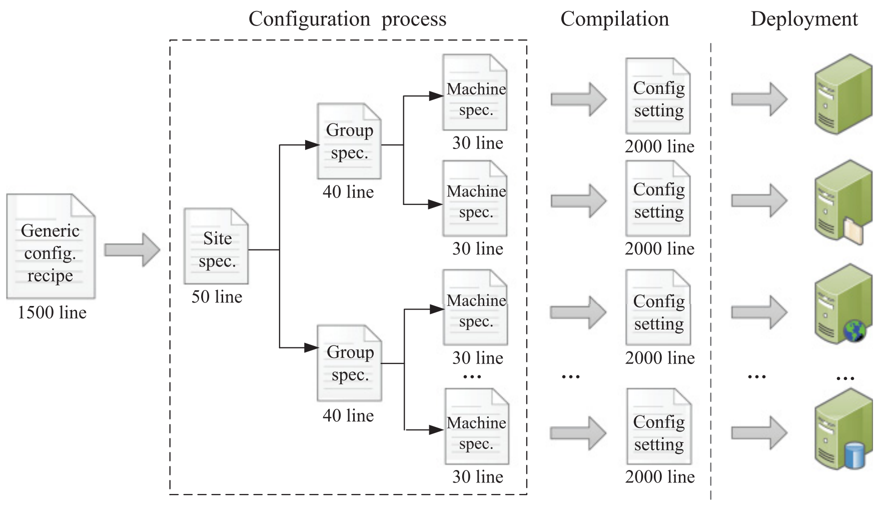


Fig. 4. A demonstrative example of automatic configuration deployment processes using object-oriented declarative languages.

Flight Data Recorder [Verbowski et al. 2006a] from which it records all system changes, verifies approved requests, and alerts users to unknown modifications.

Oliveira et al. [2010] observe that configuration errors on one or a few machines are likely to be propagated to many others via large-scale deployment, which enlarges the impact of the problem and the subsequent repairing time. Their tool, Barricade, monitors the configuration actions and the changes of system states, and infers the expected cost if the action leads to errors. Based on the cost, it decides whether and how the system should react to the configuration changes.

10. TROUBLESHOOTING CONFIGURATION ERRORS

When configuration errors escape from the system defenses, causing failures, the users have to troubleshoot the errors based on the failure symptoms. Due to the lack of effective and efficient tool support (cf. Section 3.4), troubleshooting configuration errors mostly relies on manual efforts involving examining the configuration settings, analyzing log messages, checking system states, and searching knowledge base articles. If the users cannot successfully diagnose the problem by themselves, they have to call technical support for assistance. This often falls into more costly diagnosis cycles, including collecting site information and remote debugging. Therefore, it is highly desired to have automatic solutions and tool support to facilitate the troubleshooting process based on the failure-site information (e.g., log messages, stack traces, and core dumps).

As discussed in Section 3.4, troubleshooting configuration errors has to address the following two challenges. First, since the users may not have debugging information (e.g., source code) or the expertise of debugging complex systems, troubleshooting cannot follow the similar interactive process as software debugging, which asks users to examine the program runtime execution and reason out the root causes (e.g., using GDB). Second, the goal of misconfiguration troubleshooting is to pinpoint the erroneous configuration settings rather than the bugs inside the software. Thus, existing diagnosis support (e.g., reconstructing the execution paths and reproducing the failures) may not be sufficient or useful for helping users find the configuration errors. In other words, troubleshooting configuration errors needs to extend from errors in the program to include errors in the configuration.

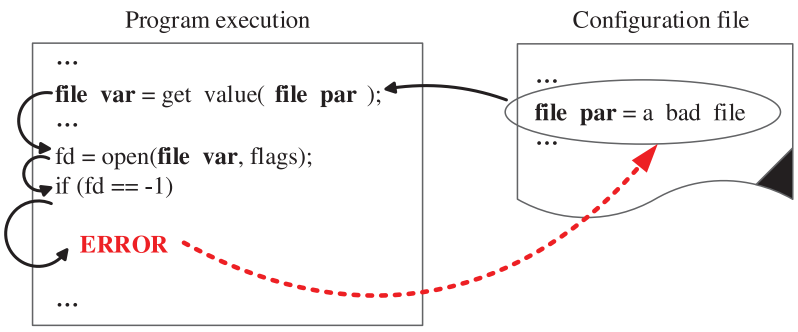


Fig. 5. An example of misconfiguration troubleshooting based on the causality analysis of runtime execution. The black, solid lines represent the execution (which is either replayed or reconstructed by the troubleshooting tool), while the red, dotted line represents the causal dependency between the symptom and the failure.

10.1. Causality Analysis

There are several troubleshooting approaches that attempt to automatically identify the erroneous setting as root causes of the failures. They do this by analyzing the causal relationships between the settings and the failures. From the system's perspective, configuration settings are one type of system inputs that are read and stored in the system variables, and then used during the runtime execution. As they deviate from the normal executions, configuration errors lead to the *failure executions*, that is, executions ended with system failures. Causality analysis attempts to reason out the causality between the configuration settings (cause) and the failure execution (effect).

Figure 5 demonstrates the process of troubleshooting configuration errors using causality analysis. First, the troubleshooting tool needs to record or reconstruct the program execution that starts from reading the configuration files and ends with the failure. In this example, the execution (represented as the black, solid line) first reads the configuration file and stores the settings of configuration parameters in the program variables (e.g., the `file_par` parameter is stored in the `file_var` variable). Then, it uses the erroneous setting during the execution and triggers the failure (`ERROR`). The troubleshooting tool analyzes the failure execution, and automatically reasons out the causality between `file_par` and the `ERROR`. Based on this information, the tool reports that the failure is caused by the configuration error (the bad setting of `file_par`) in the configuration file.

ConfAid [Attariyan and Flinn 2010, 2011] is a representative misconfiguration troubleshooting tool based on causality analysis. To troubleshoot a configuration error, ConfAid first instruments the program binaries to insert the monitor code, which will record the information flow during the program execution. Then, it reruns the program in order to reproduce the failures using the instrumented binaries. Based on the recorded execution information, ConfAid identifies the causal dependencies between each configuration setting and the failure. The configuration setting with stronger dependency to the failure is more likely to be the root cause of the failure.

On top of ConfAid, the authors further design and implement X-ray [Attariyan et al. 2012], a tool specialized for troubleshooting configuration errors that cause performance anomalies. X-ray records the performance summarization of each configuration setting during the program execution, quantifying the performance impact caused by the setting. The settings with the bigger impact are considered to have stronger causality with the performance anomaly. ConfAid and X-ray use a lightweight monitoring mechanism so that a configuration problem can be captured online with low overhead and reproduced offline for the causality analysis.

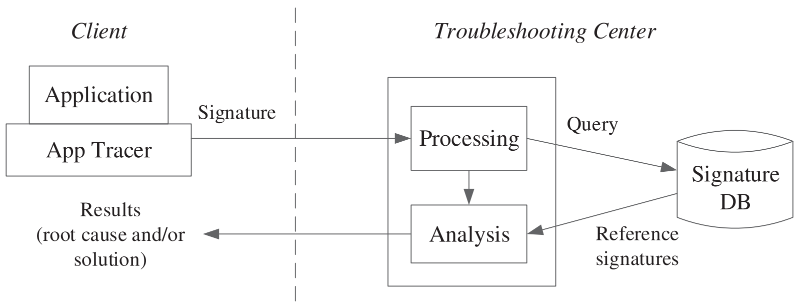


Fig. 6. The work-flow of signature-based troubleshooting. The client-side App Tracker generates and sends the signature of the sick system/execution to the troubleshooting center. The troubleshooting tool automatically reasons out the errors (or solutions) by analyzing the signature with the reference signatures.

Different from ConfAid, ConfAnalyzer [Rabkin and Katz 2011a] takes a static approach that precomputes the causal dependencies between the configuration parameters and the potential failure points. Thus, it does not require instrumenting the binaries and reproducing the failures. ConfAnalyzer analyzes the dataflow and control flow of each configuration parameter on each potential execution path in the program. It stores the mapping from the potential failure paths to the configuration parameters in a table. To diagnose a failure, ConfAnalyzer first tries to reconstruct the failure execution path based on log messages and stack traces; then it queries the table using the failure path to obtain the dependent configuration parameters whose settings are suspected to be erroneous.

10.2. Signature-Based Approaches

Signature-based troubleshooting approaches attempt to automatically reason out the configuration errors by comparing the signature of the failure execution with the reference signatures.[12] Reference signatures are usually recorded from executions with known system behavior: they can be either normal executions or the known failure executions recorded from previous troubleshooting efforts. Figure 6 demonstrates the troubleshooting using the signature-based approach. As such approach requires a large number of signatures (usually collected from a large number of system instances in real uses), troubleshooting based on signature-based analysis is mostly adopted by the companies with a large user base such as Microsoft.

PeerPressure [Wang et al. 2004] and its predecessor Strider [Wang et al. 2003] compare the failure signature with signatures of normal executions.[13] Upon a failure, the client-side App Tracker captures the Windows Registry entries as the signature of the failure execution and sends it to PeerPressure. These Registry entries are treated as misconfiguration suspects. Next, PeerPressure queries a centralized signature database and fetches the respective settings of the entries that were collected from the sample machines. Then, it uses Bayesian estimation to calculate the probability of an entry to be erroneous and outputs the ranks of every entry. The PeerPressure

[12]A *signature* refers to the information recorded at the system runtime. Its semantics depend on the signature collector of the troubleshooting approach. For example, the signature could be the configuration settings [Wang et al. 2003], system call traces [Yuan et al. 2006], and dependency set [Attariyan and Flinn 2008].

[13]PeerPressure assumes that an application would function correctly on most of the deployed machines; therefore, it uses the information distilled from a large enough set of sample machines as the "statistical golden states." In this way, they can save the manual effort of labeling whether a machine is "healthy" or "sick" (the labeling is required in Strider).

technology has been transferred into the Microsoft Product Support Services (PSS) toolkit.

Different from PeerPressure, Yuan et al. [2006] propose to compare the signature of the failure execution with other known failure signatures. If a match is found, the previous troubleshooting efforts can be reused to address the current configuration error. In their proposed tool, the App Tracker collects system call traces as the signature at the client side. Support Vector Machine (SVM) classification algorithms are used to match the signature with the category generated from the signatures of known problems. Then, the solution of the known problem will be returned to users. Attariyan and Flinn [2008] show that the *dependency set* of a process is robust and accurate signatures across variations of Unix operating systems (the dependency set is defined as the set of objects read by the process and its descendants during their execution such as files, directory entries, devices, etc).

The idea of signature-based analysis has also been applied by several other troubleshooting tools in different domains. NetPrints [Aggarwal et al. 2009] applies decision tree-based learning methods on working and nonworking configuration snapshots to generate signatures that distinguish different models of failures. It identifies home-network misconfigurations if the error signatures match known failure signatures. ConfDiagnoser [Zhang and Ernst 2013] instruments the bytecode of Java programs to insert *predicates* (code capturing the program state) and records the runtime outcome of every predicate as the signatures. It matches the signature with previous signatures based on their cosine similarity, and then infers the configuration errors according to the predicates.

10.3. Checkpoint-Based Approaches

Checkpointing is a technique of recording the history of the system states in a time series. It allows users (e.g., system administrators) to check how the system transitions from the working state to the failure state when a system failure occurs. To troubleshoot the configuration error that causes the failure, the administrator can compare the state of the system immediately before and after the failure using the UNIX `diff` command [Whitaker et al. 2004].

Chronus [Whitaker et al. 2004] automates the search for the time when the system transitions from a working state to a failure state. It only checkpoints the persistent storage based on a time-travel disk that has relatively low overhead compared with whole-system checkpointing. Chronus uses a virtual machine monitor to instantiate, boot, and test historical snapshots of the system in order to determine if a system snapshot represents a working state. Snitch [Mickens et al. 2007] leverages the Flight-Data Recorder [Verbowski et al. 2006a] to construct the timeline views of the system states. The time lines are useful for validating the root cause hypothesis that the decision tree-based troubleshooting procedures suggest.

11. OPEN PROBLEMS

Configuration errors are not newly emerged problems. They have been associated with the system since the first day the configuration is exposed. However, in recent years, configuration errors have become particularly prevalent and severe with the following trends.

(1) *The systems are becoming more complex and larger in scale*. The scale of today's systems has increased dramatically. The notion of *large-scale* systems a decade ago mainly refers to clusters consisting of tens of machines, while today's large-scale systems (such as the cloud and data-center systems) could run on top of tens of thousands of heterogeneous, geodistributed computing nodes with a variety of corunning software components. Such system scale significantly enlarges the impact of configuration

errors. First, the configuration errors on critical code paths or core system components may affect the entire system. For example, the Google outage in 2012 was caused by the misconfigured load balancing servers that affected other components including sync and quota servers [Brodkin 2012]. Second, one configuration error could be rolled out to hundreds, or even thousands, of system instances, which significantly enlarges its impact.

Moreover, the complexity of today's systems has increased tremendously. Many of today's systems contain multiple components with sophisticated dependencies and interactions. For example, the GoogleDocs services rely on around 50 other services inside Google [Wilkes 2011]. A single configuration error in one component might cause catastrophe through cascading effects.

The scale and complexity of the systems also increase the difficulties of configuration. Since it is prohibitively hard for a single administrator to be in charge of the entire system, large-scale systems are usually configured by multiple administration teams. Each team is responsible for one component or one aspect of the system. However, the lack of global view and control of the entire system significantly increases the error proneness of configuration. Many of the existing approaches and tools designed for an individual component are not so appropriate in this case. In fact, it is even unrealistic to expect the diverse components to adopt any kind of unified configuration process—the real configuration problem is in integrating and configuring the connections between them.

(2) *The systems are becoming more accessible and affordable to users.* With the proliferation of open-source software as well as the economic, on-demand pricing models of cloud computing [Armbrust et al. 2009], today's systems (including server and infrastructure systems) are no longer monopolized by large enterprises. Many small companies or even end users are able to maintain and operate their own systems [Andreessen 2011] such as LAMP-based Web servers, Hadoop clusters, and OpenStack cloud systems. The prevalent usage of systems leads to the prevalence of configuration errors, as more and more system administrators are not professional personnel.

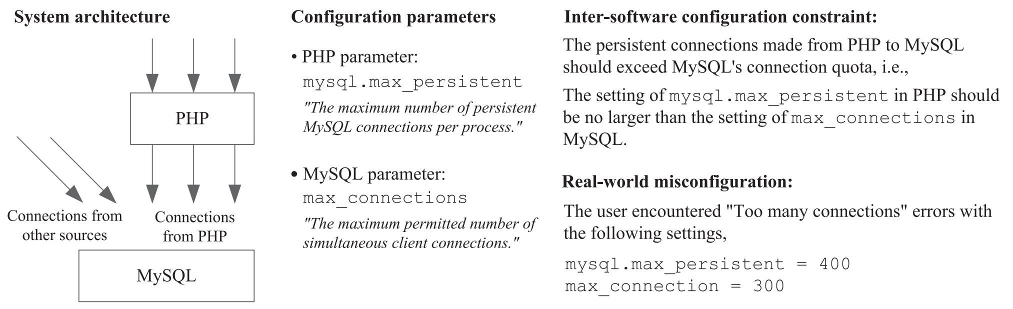
We believe that these trends will continue in the next decades. Thus, it is time to further develop approaches and practices to tackle configuration errors. In this section, we discuss some open problems with regard to system configuration and the challenges of attacking the problems. We believe that the breakthrough of these problems leads to the eventual steps toward addressing configuration problems, with the hope that the discussion could inspire follow-up research and practices.

11.1. Rethinking and Redesigning Configuration

The prevalence of today's configuration difficulties and errors, as well as the resulting severe impact on system availability, reveals the importance of rethinking and redesigning configuration. We advocate that configuration design should aim at proactively eliminating users' configuration errors in the first place, rather than passively detecting and diagnosing after the errors occur (which is inefficient and costly). Reducing error proneness and configuration complexity should have the same priority as feature richness, time to market, and performance. Indeed, in the landmark study of system reliability based on Tandem computers, Gray [1985] notes,

> "*The top priority for improving system availability is to reduce administrative mistakes by making self-configured systems with minimal maintenance and minimal operator interaction. Interfaces that ask the operator for information or ask him to perform some function must be simple, consistent and operator fault-tolerant.*"

There have been a few research proposals advocating to rethink and redesign software systems to address operation/administration errors and cost, including the RISC-style



Configuration parameters

- PHP parameter:
  `mysql.max_persistent`
  "The maximum number of persistent MySQL connections per process."
- MySQL parameter:
  `max_connections`
  "The maximum permitted number of simultaneous client connections."

Inter-software configuration constraint:

The persistent connections made from PHP to MySQL should exceed MySQL's connection quota, i.e.,

The setting of `mysql.max_persistent` in PHP should be no larger than the setting of `max_connections` in MySQL.

Real-world misconfiguration:

The user encountered "Too many connections" errors with the following settings,

```
mysql.max_persistent = 400
max_connection = 300
```

Fig. 7. A real-world example of a cross-component configuration error in a LAMP-based Web server.

system design [Chaudhuri and Weikum 2000], recovery-oriented computing [Brown and Patterson 2001], autonomic computing [Kephart and Chess 2003], and human-aware systems [Bianchini et al. 2005]. Unfortunately, few of them are specialized for configuration or provide solutions that address the challenges of tackling configuration errors; thus, it is hard for the stakeholder to take actions.

We believe that the key to configuration design and implementation is to follow the user-centric design philosophy, with the goal of simplifying configuration complexity and reducing error proneness. This requires system vendors to build user-feedback loops to understand users' configuration practices, difficulties, and common mistakes. Benson et al. [2009] develop a suite of complexity models and metrics to describe the routing configuration of a network. These models are designed to align with the complexity of mental model operators use when reasoning about the networks, including the complexity behind configuring network routers, the complexity arising from identifying and defining distinct roles for routers, and the inherent complexity of the network policies to be implemented. With these models, the network operators are able to compare different configuration solutions and understand the complexity and overhead of managing the configured network [Benson et al. 2011]. Xu et al. [2015] make one of the first steps to study users' configuration settings in the field. They find that many configurations are overdelivered: they are seldom leveraged by users but unnecessarily increase the complexity. They propose to reduce the unnecessary complexity to balance the simplicity (usability) and flexibility (configurability). We advocate that such user-centric design practices should be applied to all kinds of systems that require intensive configuration, in a similar way to how developers understand the users' problems for UI/UX design. After all, configuration itself is a type of system interface that is more challenging and critical to use.

11.2. Dealing with Cross-Component Configuration

Most of the existing approaches and tools (discussed in Sections 8–10) are designed for configuration in a single software component; they are not able to tackle configuration errors that break the cross-component dependencies and correlations. Unfortunately, such configuration errors are not uncommon. Yin et al. [2011] report that "*although most misconfiguration is located within each examined application, still a significant portion (21.7%–57.3%) of cases involve configuration beyond the application itself or span across multiple hosts.*" Figure 7 demonstrates a real-world example of a configuration error that violates the value dependency of two configuration parameters, one from PHP and the other from MySQL. Welsh [2013] describes a crash case in Google where the configuration changes of one system component (generating more socket connections) caused failures of another system component.

The key challenge of cross-component configuration is to understand the inexplicit interactions related to configuration among different system components in a large-scale, complex system. As exemplified by Figure 7, the configuration settings could work perfectly well for each software component, but combining them leads to errors due to the interactions among components. Unfortunately, information of such interactions is often inexplicit and poorly understood. The situation is worsened with system federation where different components are maintained by different teams or organizations that may not collaborate with each other. One of the real configuration problems is in integrating and configuring the connections between the components.

In particular, cross-component configuration errors are difficult to troubleshoot if the users or support engineers are not aware of the hidden interactions. In the Google's case [Welsh 2013], "*the engineer on call spent many, many hours trying different things and trying to isolate the problem, without success.*" It is highly desired to have new approaches and tools that can tackle configuration errors by analyzing the entire software stacks as a whole system. A few existing approaches make efforts to troubleshoot configuration errors across multiple components, for example, to track the interprocess information flows of users' configuration settings, ConfAid (cf. Section 10.1) instruments the system calls that create sockets and pipes to intercept messages with special headers. Rabkin and Katz [2011b] manually annotate known RPC calls in Hadoop's source code in order to track the dataflows of configuration parameters. Unfortunately, these approaches do not scale for large-scale, long-run systems and are limited in the analysis they can perform.

11.3. Facilitating Configuration Collaboration

Configuring large-scale, federated systems often requires collaborations between system administrators and operators across multiple teams or even multiple organizations. Without proper collaborative procedures, configuration would be prone to errors and misinterpretation. Unfortunately, not much attention has been paid to provide solutions and tool support to facilitate the configuration collaboration. Unlike software development, few configuration tools enable peer review, accounting (tracking who is responsible for which configuration settings), and configuration error tracking. We believe that more research efforts and practices should be conducted to enforce standardized collaborative procedures for configuration.

There are several projects that look at the collaboration aspects and build tooling support to assist the procedures. Vanbrabant et al. [2011] integrate fine-grained access control on top of Puppet to enforce configuration changes to go through the review process by peer administrators or managers. It shares the same benefits as code review, including more eyeballs on bugs, style compliance, knowledge sharing, etc. Most importantly, it ensures the agreements of different teams and organizations. Anderson and Cheney [2012] propose tracking the provenance of each parameter setting (including where-provenance, dependency tracking, and override history) for configuration collaborations. The provenance information can identify who has contributed to the parameter setting and in what way. PRESTO [Enck et al. 2007, 2009] aims at enabling *modularity*, a software-engineering practice, for network configuration. It automatically constructs a router-native configuration by gluing together information sources of myriad organizations. With PRESTO, domain experts can focus on codifying configuration templates, while network operators initialize and turn-up the templates using router- and environment-specific information.

11.4. Providing Language Support for Configuration

Providing language support is a promising direction of tackling configuration problems. As discussed in Section 6.1, declarative configuration languages can make configuration easy and efficient by expressing the high-level actions and masking low-level minutiae of settings. In addition, better language support can potentially improve or enforce the expressiveness, modularity, and verifiability of configuration to eliminate configuration errors. For example, the type-based configuration errors (e.g., typos, numeric overflow/underflow) can be easily detected by language-level type checking.

On the other hand, the configuration language should be carefully designed, since the abuse of using programming languages for configuration may impair the provability, security, and usability [Mason 2011]. The following quotation shows the usability impairment of using Perl as configuration language [Walkin 2012]:

> "*In a very short time the lure of unused expressiveness of such Turing-complete environment prevailed and people started to write for-loops around data pieces and doing other tricks to remove redundancy from the configuration. It turned out to be a disaster in the end, with configuration becoming unmaintainable and flaky.*"

The research of designing the "right" language support to enforce good system configuration practices is important. The language should be simple and manageable (as the plain texts), while remaining expressive, provable, and verifiable (as programming languages). The Click configuration language [Kohler et al. 2000] is one of the good examples. It provides a modular software router framework where users configure routing decisions, dropping/scheduling policies, and packet manipulations using a simple, declarative dataflow language. This language is amenable to dataflow analysis and other conventional programming language technologies. Applied to a router configuration, these technologies can achieve high-level results, for example, optimizing the router or verifying its high-level properties. We envision the design and implementation of more programming language support for other domain-specific configuration tasks.

11.5. Improving Configuration UIs

Another direction of facilitating users' configuration tasks and reducing their mistakes is to design efficient and user-friendly configuration UIs. The configuration UIs have been completely overlooked in past decades. Currently, the *de facto* configuration UI is still the text file, which is not able to provide guidance, information, or feedback. The assumption of using files as the UI is that users know exactly what and how to configure so that they can directly edit the entries in the file. However, this assumption does not hold. Hubaux et al. [2012] survey 90 Linux and eCos users and report that many respondents complained about the lack of guidance and the low quality of the advice provided by the configuration UI. Specifically, enabling an inactive configuration parameter can be difficult, because "*advice is often incomplete, hard to understand, or even incorrect.*"

Building an informative and handy configuration UI is not trivial, and needs support from the systems level. As pointed out by Oppenheimer et al. [2003], a good configuration UI is not equivalent to simple graphic wrappers around existing configuration files or the command-line interfaces. Instead, configuration UI should help users understand the configuration logic, their constraints and dependencies, as well as the potential impact of the configuration settings to the entire system. For example, Maxion and Reeder [2005] observe that a significant percentage of misconfiguration of NTFS permissions are because of the configuration UI not providing adequate information to users. A new UI design that presents all task-relevant information and allows status checking can effectively reduce configuration errors of NTFS permissions by 94%.

In fact, a number of system methods have been proposed to automatically, systematically extract a variety of types of configuration information, including constraints, dependencies, and correlations (e.g., Kiciman and Wang [2004], Ramachandran et al. [2009], Nadi et al. [2014], Zhang et al. [2014], and Xu et al. [2013]); however, little of the information has been integrated into the configuration UIs. We hope that future configuration UIs could visualize and present a rich set of configuration information to facilitate users during the configuration processes.

11.6. Understanding Configuration Problems in the Cloud

The wide adoption of the *cloud computing* principle has posed a number of new challenges of configuration due to the scale of the systems, the complexity caused by the component interactions, and the dynamics of the systems [Reiss et al. 2012]. For example, Zhang and Liu [2011] report that system migration (from a local data center to the cloud, or from one data center to another data center inside the cloud) is a typical source of configuration errors in cloud systems.

The configuration errors in cloud systems often have big impact, as the cloud systems often serve as the back-end and platform support for many client-side applications and services. In addition, the impact of one configuration error would be significantly enlarged, as the erroneous configuration settings are often replicated onto a large number of system instances. Furthermore, configuration errors in the cloud systems are hard to diagnose due to the system scale and complex environment. Therefore, it poses high requirements for the configuration design and implementation of the cloud systems, as well as the practices of managing and operating these systems (including the deployment, monitoring, refinement, etc.).

Several recent studies investigate the software systems commonly deployed in the cloud. Gunawi et al. [2014] conduct a one-year study of development and deployment issues of six popular cloud systems, including MapReduce, HDFS, HBase, Cassandra, ZooKeeper, and Flume, based on the JIRA cases. They report that configuration issues are the fourth largest category (14%) including both wrong configuration settings and inappropriate settings. Yuan et al. [2014] analyze the production failures in a similar set of cloud-based software systems and report that 23% of the failures are caused by configuration changes, among which 30% have configuration errors and the remaining majority involve valid changes. Based on the results, they suggest extending existing testing tools to combine configuration changes with other operations.

Unfortunately, there is limited understanding of the operational practices and the configuration problems of industrial cloud systems in the research community. We encourage the cloud vendors to open and share their practices and experiences in managing configuration in the cloud systems.

11.7. Helping End-User Administrators

With the proliferation of free software and economic infrastructure support, we can envision that more and more nontechnical professions will start to configure and manage their own systems to support their work or hobbies. Similar to the term "end-user programmer" [Ko et al. 2011], we use "end-user system administrators" to differentiate them from their professional counterparts based on their goals: professionals are paid to maintain systems over time; end users in contrast configure and use systems to support some goals in their own domains of expertise.

Unlike professional administrators or technicians, end-user administrators may not have sufficient experience or background to accomplish configuration tasks; some of them even have difficulties in understanding the configuration semantics and terms. The following quote gives one example of the user's difficulties in understanding the configuration semantics documented in the user manual. It is from a user's email sent to the developer in OpenLDAP mailing list [2004] in response to the developer's comment that "*The reference manual already states, near the top...*"

"You are assuming that those who read that, understood what the context of 'user' was. I most assuredly did not until now. Unfortunately, many of us don't come from UNIX backgrounds and though pick up on many things, some things which seem basic to you guys elude us for some time."

More attention should be paid on end-user system configuration and administration to empower end users to manage their own systems. Specialized configuration design and support for end-user administrators are required. Such design and support should focus more on helping users enable functionalities and features rather than fine tune-up of other aspects (e.g., performance, availability, security, etc.), because end users mostly want convenient, immediate solutions to achieve their short-term goals. Like the technical books written for dummies (e.g., Coar [1998]), the "dummy" version of configuration should be designed and integrated with less assumption on the users' abilities and more guidance and feedback for users.

11.8. Awareness and Responsibility

One psychological reason behind the prevalence of today's configuration errors lies in the developers' attitudes toward these errors. Unlike software bugs for which developers take full responsibility and active roles, many developers take a laid-back role in dealing with configuration errors because "*they are not bugs but users' incorrect settings*" [Xu et al. 2013]. The implication is that misconfiguration is the users' faults, not the developers' business. Such an attitude is reflected in at least the following two aspects: (1) configuration errors are much less rigorously tracked; and (2) after configuration errors are identified as the root causes of system failures, developers often do not take further action to *fix* the error, such as changing the code or releasing patches. Xu et al. [2013] share their experience of interacting with developers to fix system vulnerabilities to configuration errors (cf. Section 7) and report certain typical thinking of developers, including assuming that users read the manuals carefully, that open-source users read the code, and even that administrators do not make mistakes.

It is critical for developers to be aware that actively handling configuration errors benefits not only users but also themselves. *Unlike software bugs, most configuration errors can be easily fixed by users if the system can react gracefully to users' configuration errors.* If a user's misconfiguration causes the system to crash, hang, or fail silently, the user has no choice but report to technical support. Not only do the users suffer from system downtime, but also the developers, who have to spend time and effort troubleshooting the errors and perhaps compensating the users' losses. On the other hand, if the system can deny the error and print explicitly log messages, the users can directly fix the mistakes by themselves. Furthermore, handling configuration errors also significantly eases developers' troubleshooting efforts. Yin et al. [2011] report that good system feedback with explicit messages can shorten the diagnosis time by 3–13 times compared with the cases with ambiguous messages, and by 1.2–14.5 times compared with the cases with no message.

Taking an active role represents being aware of the importance and the responsibility of handling configuration errors. It spins across the entire lifecycles of the software systems. We wish that with more education of the severity and the cost of configuration errors, configuration problems will no longer lie in the gray zone between the developers and the users.

12. CONCLUSION

In this article, we have discussed the characteristics of configuration errors, the challenges of tackling configuration errors, and the state-of-the-art systems approaches that deal with different types of configuration errors. The surveyed approaches span across a variety of system development stages, including design, implementation, and quality assurance. We hope that the discussion can help system vendors (including architects, developers, and testing engineers) build user-friendly configuration so as to eliminate configuration errors in the first place. The article also covers approaches to attacking configuration errors in different usage scenarios, including detection and validation, deployment and management, and failure diagnosis. We hope that it can serve as a systematic exploration for users (especially system administrators) and technical-support engineers to leverage existing technologies to actively address configuration problems. Last but not least, we have discussed the open problems with regard to system configuration. We encourage further research and practices to address these problems toward the escape from configuration errors.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments that help improve the article. We also thank Geoffrey Voelker and Alex Snoeren for providing feedback for the early draft, and Robbert van Renesse for helping prepare the final version of the article. We are grateful to Shankar Pasupathy, Ryan Huang, Jiaqi Zhang, Xuepeng Fan, Tianwei Sheng, Weiwei Xiong, Xinxin Jin, Long Jin, Rishi Kapoor, Danny Huang, Soyeon Park, Vineet Pandey, Ding Yuan, Shan Lu, Geoffrey Voelker, Stefan Savage, Alex Snoeren, Scott Klemmer, and Bill Griswold for the discussions that help us understand different perspectives of the problem. We also thank Fei Ren and Edith Yuh for the paper proofreading. The article is based on Tianyin Xu's report for the Ph.D. Research Examination at University of California, San Diego. Tianyin Xu thanks Mihir Bellare, Alex Snoeren, and Geoffrey Voelker for their presence on his examination committee.

REFERENCES

Bhavish Aggarwal, Ranjita Bhagwan, Tathagata Das, Siddharth Eswaran, Venkata N. Padmanabhan, and Geoffrey M. Voelker. 2009. NetPrints: Diagnosing home network misconfigurations using shared knowledge. In *Proceedings of the 6th USENIX Symposium on Networked System Design and Implementation (NSDI'09)*.

Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The stratosphere platform for big data analytics. *The International Journal on Very Large Data Bases (VLDBJ)* 23, 6 (December 2014), 939–964.

Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. 2002. Hippodrome: Running circles around storage administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*.

Paul Anderson. 1994. Towards a high-level machine configuration system. In *Proceedings of the 8th USENIX System Administration Conference (LISA-VIII)*.

Paul Anderson and James Cheney. 2012. Toward provenance-based security for configuration languages. In *Proceedings of the 4th USENIX Workshop on the Theory and Practice of Provenance (TaPP'12)*.

Paul Anderson and Edmund Smith. 2005. Configuration tools: Working together. In *Proceedings of the 19th Large Installation System Administration Conference (LISA'05)*.

Marc Andreessen. 2011. Why software is eating the world. *The Wall Street Journal* (August 2011).

Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2009. *Above the Clouds: A Berkeley View of Cloud Computing*. Technical Report UCB/EECS-2009-28. University of California Berkeley.

Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*.

Mona Attariyan and Jason Flinn. 2008. Using causality to diagnose configuration bugs. In *Proceedings of 2008 USENIX Annual Technical Conference (USENIX ATC'08)*.

Mona Attariyan and Jason Flinn. 2010. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*.

Mona Attariyan and Jason Flinn. 2011. Automating configuration troubleshooting with ConfAid. *USENIX ;login:* 36, 1 (February 2011), 27–36.

Rob Barrett, Eser Kandogan, Paul P. Maglio, Eben Haber, Leila A. Takayama, and Madhu Prabaker. 2004. Field studies of computer system administrators: Analysis of system management tools and practices. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work (CSCW'04)*.

Luiz André Barroso and Urs Hölzle. 2009. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers.

Lujo Bauer, Scott Garriss, and Michael K. Reiter. 2011. Detecting and resolving policy misconfigurations in access-control systems. *ACM Transactions on Information and System Security (TISSEC)* 14, 1 (May 2011), 1–28.

Theophilus Benson, Aditya Akella, and David Maltz. 2009. Unraveling the complexity of network management. In *Proceedings of the 6th USENIX Symposium on Networked System Design and Implementation (NSDI'09)*.

Theophilus Benson, Aditya Akella, and Aman Shaikh. 2011. Demystifying configuration challenges and trade-offs in network-based ISP services. In *Proceedings of 2011 Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'11)*.

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (February 2010), 66–75.

Ricardo Bianchini, Richard P. Martin, Kiran Nagaraja, Thu D. Nguyen, and Fabio Oliveira. 2005. Human-aware computer system design. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS X)*.

Charles Border and Kyrre Begnum. 2014. Educating system administrators. *USENIX ;login:* 39, 5 (October 2014), 36–39.

Jon Brodkin. 2012. Why Gmail Went Down: Google Misconfigured Load Balancing Servers. Retrieved from http://arstechnica.com/information-technology/2012/12/why-gmail-went-down-google-misconfigured-chromes-sync-server/.

Aaron Brown. 2001. *Towards Availability and Maintainability Benchmarks: A Case Study of Software RAID Systems*. Technical Report UCB//CSD-01-1132. University of California, Berkeley.

Aaron B. Brown and David A. Patterson. 2001. To err is human. In *Proceedings of the 1st Workshop on Evaluating and Architecting System Dependability (EASY'01)*.

Aaron B. Brown and David A. Patterson. 2003. Undo for Operators: Building an undoable e-mail store. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX ATC'03)*.

Mark Burgess. 1995. A site configuration engine. *USENIX Computing Systems* 8, 3 (1995), 309–337.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*..

Jeffery S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. 2001. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*.

Surajit Chaudhuri and Gerhard Weikum. 2000. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'00)*.

Kai Chen, Chuanxiong Guo, Haitao Wu, Jing Yuan, Zhenqian Feng, Yan Chen, Songwu Lu, and Wenfei Wu. 2010. Generic and automatic address configuration for data center networks. In *Proceedings of 2010 Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'10)*.

Ken A. L. Coar. 1998. *Apache Server For Dummies*. IDG Books Worldwide Inc.

Computing Research Association. 2003. *Grand Research Challenges in Information Systems.* Technical Report. Retrieved from http://archive.cra.org/reports/gc.systems.pdf.

Thomas Delaet, Wouter Joosen, and Bart Vanbrabant. 2010. A survey of system conguration tools. In *Proceedings of the 24th Large Installation System Administration Conference (LISA'10)*.

Narayan Desai, Rick Bradshaw, Scott Matott, Sandra Bittner, Susan Coghlan, Remy Evard, Cory Lueninghoener, Ti Leggett, John-Paul Navarro, Gene Rackow, Craig Stacey, and Tisha Stacey. 2005. A case study in configuration management tool deployment. In *Proceedings of the 19th Large Installation System Administration Conference (LISA'05)*.

John DeTreville. 2005. Making system configuration more declarative. In *Proceedings of the USENIX 10th Workshop on Hot Topics in Operating Systems (HotOS-X)*.

Eelco Dolstra and Armijn Hemel. 2007. Purely functional system configuration management. In *Proceedings of the USENIX 11th Workshop on Hot Topics in Operating Systems (HotOS-XI)*.

Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with ituned. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'09)*.

Kevin Elphinstone and Gernot Heiser. 2013. From L3 to seL4: What have we learnt in 20 years of L4 microkernels?. In *Proceedings of the 24th Symposium on Operating System Principles (SOSP'13)*.

William Enck, Patrick McDaniel, Subhabrata Sen, Panagiotis Sebos, Sylke Spoerel, Albert Greenberg, Sanjay Rao, and William Aiello. 2007. Configuration management at massive scale: System design and experience. In *Proceedings of 2007 USENIX Annual Technical Conference (USENIX ATC'07)*.

William Enck, Thomas Moyer, Patrick McDaniel, Subhabrata Sen, Panagiotis Sebos, Sylke Spoerel, Albert Greenberg, Yu-Wei Eric Sung, Sanjay Rao, and William Aiello. 2009. Configuration management at massive scale: System design and experience. *IEEE Journal on Selected Areas in Communications (JSAC)* 27, 3 (April 2009), 323–335.

Nick Feamster and Hari Balakrishnan. 2005. Detecting BGP configuration faults with static analysis. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)*.

Patrick Goldsack, Julio Guijarro, Steve Loughran, Alistair Coles, Andrew Farrell, Antonio Lain, Paul Murray, and Peter Toft. 2009. The smartfrog configuration management framework. *SIGOPS Operating System Review* 43 (January 2009), 16–25.

Jim Gray. 1985. Why do computers stop and what can be done about it? *Tandem Technical Report 85.7* (June 1985).

Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC'14)*.

Eben M. Haber and John Bailey. 2007. Design guidelines for system administration tools developed through ethnographic field study. In *Proceedings of the 2007 ACM Conference on Human Interfaces to the Management of Information Technology (CHIMIT'07)*.

Herodotos Herodotou and Shivnath Babu. 2011. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. In *Proceedings of the 37th International Conference on Very Large Data Bases (VLDB'11)*.

John A. Hewson, Paul Anderson, and Andrew D. Gordon. 2012. A declarative approach to automated configuration. In *Proceedings of the 26th Large Installation System Administration Conference (LISA'12)*.

Dennis G. Hrebec and Michael Stiber. 2001. A survey of system administrator mental models and situation awareness. In *Proceedings of the 2001 ACM SIGCPR Conference on Computer Personnel Research (SIGCPR'01)*.

Peng Huang, William J. Bolosky, Abhishek Sigh, and Yuanyuan Zhou. 2015. KungfuValley: A systematic configuration validation framework for cloud services. In *Proceedings of the 10th ACM European Conference in Computer Systems (EuroSys'15)*.

Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. 2012. A user survey of configuration challenges in linux and ecos. In *Proceedings of 6th International Workshop on Variability Modeling of Software-intensive Systems (VaMoS'12)*.

Fabian Hueske. 2015. Juggling with Bits and Bytes. Retrieved from http://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html.

Weihang Jiang, Chongfeng Hu, Shankar Pasupathy, Arkady Kanevsky, Zhenmin Li, and Yuanyuan Zhou. 2009. Understanding customer problem troubleshooting from storage system logs. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*.

Lorenzo Keller, Prasang Upadhyaya, and George Candea. 2008. ConfErr: A tool for assessing resilience to human configuration errors. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'08)*.

Stuart Kendrick. 2012. What takes us down? *USENIX ;login:* 37, 5 (October 2012), 37–45.

Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *IEEE Computer* 36, 1 (January 2003), 41–50.

Emre Kiciman and Yi-Min Wang. 2004. Discovering correctness constraints for self-management of system configuration. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04)*.

Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)* 43, 3 (April 2011).

Eddie Kohler, Benjie Chen, M. Frans Kaashoek, Robert Morris, and Massimiliano Poletto. 2000. *Programming Language Techniques for Modular Router Configurations*. Technical Report MIT-LCS-TR-812. MIT Laboratory for Computer Science.

Nate Kushman and Dina Katabi. 2010. Enabling configuration-independent automation by non-expert users. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*.

Craig Labovitz, Abha Ahuja, and Farnam Jahanian. 1999. Experimental study of internet stability and backbone failures. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS'99)*.

Jean-Claude Laprie. 1995. Dependable computing: Concepts, limits, challenges. In *Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing (FTCS'95)*.

William LeFebvre and David Snyder. 2004. Auto-configuration by file construction: Configuration management with newfig. In *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA'04)*.

Lim Yan Liang. 2013. Linkedin.com inaccessible on Thursday because of server misconfiguration. Retrieved from http://www.straitstimes.com/breaking-news/singapore/story/linkedincom-inaccessible-thursday-because-server-misconfiguration-2013.

Cory Lueninghoener. 2011. Getting started with configuration management. *USENIX ;login:* 36, 2 (April 2011), 12–17.

Ratul Mahajan, David Wetherall, and Tom Anderson. 2002. Understanding BGP misconfiguration. In *Proceedings of 2002 Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'02)*.

Justin Mason. 2011. Against The Use of Programming Languages in Configuration Files. Retrieved from http://taint.org/2011/02/18/001527a.html. (2011).

Roy A. Maxion and Robert W. Reeder. 2005. Improving user-interface dependability through mitigation of human error. *International Journal of Human-Computer Studies* 63, 1-2 (July 2005), 25–50.

Paul McNamara. 2009. Missing dot drops Sweden off the Internet. Retrieved from http://www.computerworld.com/article/2529287/networking/opinion–missing-dot-drops-sweden-off-the-internet.html.

James Mickens, Martin Szummer, and Dushyanth Narayanan. 2007. Snitch: Interactive decision trees for troubleshooting misconfigurations. In *Proceedings of the 2nd USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SYSML'07)*.

Barton P. Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. 1995. *Fuzz Revisited: A Re-Examination of the Reliability of UNIX Utilities and Services*. Technical Report No. 1268. University of Wisconsin-Madison, Computer Sciences Department.

Rich Miller. 2012. Microsoft: Misconfigured Network Device Caused Azure Outage. Retrieved from http://www.datacenterknowledge.com/archives/2012/07/28/microsoft-misconfigured-network-device-caused-azure-outage/.

Rolf Molich and Jakob Nielsen. 1990. Improving a human-computer dialogue. *Communications of the ACM* 33, 3 (March 1990), 338–348.

Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining configuration constraints: Static analyses and empirical results. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*.

Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. 2004. Understanding and dealing with operator mistakes in internet services. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)*.

Sanjai Narain. 2005. Network configuration management via model finding. In *Proceedings of the 19th Large Installation System Administration Conference (LISA'05)*.

Sanjai Narain, Gary Levin, Vikram Kaul, and Sharad Malik. 2008. Declarative infrastructure configuration synthesis and debugging. *Journal of Network and System Management* 16, 3 (October 2008), 235–258.

Jakob Nielsen and Rolf Molich. 1990. Heuristic evaluation of user interfaces. In *Proceedings of the ACM CHI 90 Human Factors in Computing Systems Conference (CHI'90)*.

Donald A. Norman. 1983a. Design principles for human-computer interfaces. In *Proceedings of the ACM CHI 83 Human Factors in Computing Systems Conference (CHI'83)*.

Donald A. Norman. 1983b. Design rules based on analyses of human error. *Communications of the ACM* 26, 4 (April 1983), 254–258.

Fábio Oliveira, Kiran Nagaraja, Rekha Bachwani, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. 2006. Understanding and validating database system administration. In *Proceedings of 2006 USENIX Annual Technical Conference (USENIX ATC'06)*.

Fábio Oliveira, Andrew Tjang, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. 2010. Barricade: Defending systems against operator mistakes. In *Proceedings of the 5th ACM European Conference in Computer Systems (EuroSys'10)*.

OpenLDAP mailing list. 2004. Re: BINDDN in ldap.conf. Retrieved from http://www.openldap.org/lists/openldap-software/200407/msg00648.html. (2004).

David Oppenheimer, Archana Ganapathi, and David A. Patterson. 2003. Why do internet services fail, and what can be done about it?. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*.

Takayuki Osogami and Toshinari Itoko. 2006. Finding probably better system configurations quickly. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS / Performance'06)*.

Noam Palatin, Arie Leizarowitz, Assaf Schuster, and Ran Wolff. 2006. Mining for misconfigured machines in grid systems. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06)*.

David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. 2002. *Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*. Technical Report No. UCB//CSD-02-1175. University of California, Berkeley.

Charles Perrow. 1984. *Normal Accidents: Living with High-Risk Technologies*. Basic Books.

Ariel Rabkin and Randy Katz. 2011a. Precomputing possible configuration error diagnosis. In *Proceedings of the 26th IEEE / ACM International Conference on Automated Software Engineering (ASE'11)*.

Ariel Rabkin and Randy Katz. 2011b. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*.

Ariel Rabkin and Randy Katz. 2013. How Hadoop clusters break. *IEEE Software Magazine* 30, 4 (July 2013), 88–94.

Vinod Ramachandran, Manish Gupta, Manish Sethi, and Soudip Roy Chowdhury. 2009. Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications. In *Proceedings of the 6th International Conference on Autonomic Computing and Communications (ICAC'09)*.

James Reason. 1990. *Human Error*. Cambridge University Press.

Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC'12)*.

William F. Slater. 2002. The Internet Outage and Attacks of October 2002. Retrieved from http://www.isoc-chicago.org/internetoutage.pdf.

Yee Jiun Song, Flavio Junqueira, and Benjamin Reed. 2009. BFT for the skeptics. In *Proceedings of the Workshop on Theory and Practice of Byzantine Fault Tolerance (BFTW^3)*.

StackOverflow. 2015. 2015 Developer Survey. Retrieved from http://stackoverflow.com/research/developer-survey-2015#profile-education.

Ya-Yunn Su, Mona Attariyan, and Jason Flinn. 2007. AutoBash: Improving configuration management with operating system causality analysis. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*.

Ya-Yunn Su and Jason Flinn. 2009. Automatically generating predicates and solutions for configuration troubleshooting. In *Proceedings of 2009 USENIX Annual Technical Conference (USENIX ATC'09)*.

Keir Thomas. 2011. Thanks, Amazon: The Cloud Crash Reveals Your Importance. Retrieved from http://www.pcworld.com/article/226033/thanks_amazon_for_making_possible_much_of_the_internet.html.

Bart Vanbrabant, Joris Peeraer, and Wouter Joosen. 2011. Fine-grained access-control for the puppet configuration language. In *Proceedings of the 25th Large Installation System Administration Conference (LISA'11)*.

Nicole F. Velasquez, Suzanne Weisband, and Alexandra Durcikova. 2008. Designing tools for system administrators: An empirical test of the integrated user satisfaction model. In *Proceedings of the 22nd Large Installation System Administration Conference (LISA'08)*.

Chad Verbowski, Emre Kiciman, Arunvijay Kumar, Brad Daniels, Shan Lu, Juhan Lee, Yi-Min Wang, and Roussi Roussev. 2006a. Flight data recorder: Monitoring persistent-state interactions to improve systems management. In *Proceedings of the 7th USENIX Conference on Operating Systems Design and Implementation (OSDI'06)*.

Chad Verbowski, Juhan Lee, Xiaogang Liu, Roussi Roussev, and Yi-Min Wang. 2006b. LiveOps: Systems management as a service. In *Proceedings of the 20th Large Installation System Administration Conference (LISA'06)*.

Lev Walkin. 2012. Comment on "Why Config?" Retrieved from http://robey.lag.net//2012/03/26/why-config.html.

Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. 2004. Automatic misconfiguration troubleshooting with peerpressure. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)*.

Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang. 2003. STRIDER: A black-box, state-based approach to change and configuration management and support. In *Proceedings of the 17th Large Installation Systems Administration Conference (LISA'03)*.

Matt Welsh. 2013. What I wish systems researchers would work on. Retrieved from http://matt-welsh.blogspot.com/2013/05/what-i-wish-systems-researchers-would.html.

Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. 2004. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)*.

Wikipedia. 2014. System administrator. Retrieved from http://en.wikipedia.org/wiki/System_administrator.

John Wilkes. 2011. Ωmega: Cluster management at Google. Retrieved from https://www.youtube.com/watch?feature=player_detailpage&v=0ZFMlO98Jkct=1674s.

Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H. Xia, and Li Zhang. 2004. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th International World Wide Web Conference (WWW'04)*.

Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. 2012. Generating range fixes for software configuration. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*.

Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, you have given me too many knobs!—Understanding and dealing with over-designed configuration in system software. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*.

Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do not blame users for misconfigurations. In *Proceedings of the 24th Symposium on Operating System Principles (SOSP'13)*.

Tao Ye and Shivkumar Kalyanaraman. 2003. A recursive random search algorithm for large-scale network parameter configuration. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'03)*.

Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*.

Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. 2006. Automated known problem diagnosis with event traces. In *Proceedings of the 1st ACM European Conference in Computer Systems (EuroSys'06)*.

Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*.

Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. 2011a. Context-based online configuration error detection. In *Proceedings of 2011 USENIX Annual Technical Conference (USENIX ATC'11)*.

Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2011b. Improving software diagnosability via log enhancement. In *Proceedings of the 16th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-XVI)*.

Andreas Zeller. 2009. *Why Programs Fail: A Guide to Systematic Debugging* (2nd ed.). Morgan Kaufmann Publishers.

Gong Zhang and Ling Liu. 2011. Why do migrations fail and what can we do about it? In *Proceedings of the 25th USENIX Large Installation System Administration Conference (LISA'11)*.

Jiaqi Zhang, Lakshmi Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'14)*.

Sai Zhang and Michael D. Ernst. 2013. Automated diagnosis of software configuration errors. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*.

Sai Zhang and Michael D. Ernst. 2014. Which configuration option should I change? In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*.

Wei Zheng, Ricardo Bianchini, and Thu D. Nguyen. 2007. Automatic configuration of internet services. In *Proceedings of the 2nd ACM European Conference in Computer Systems (EuroSys'07)*.

Received August 2014; revised February 2015; accepted June 2015