Root-Cause Analysis of Service Misconfigurations in Enterprise Systems

NEGAR MOHAMMADI KOUSHKI, Computer and Information Sciences, Temple University, USA
IBRAHIM EL-SHEKEIL, Computer Science and Cybersecurity, Metro State University, USA
KRISHNA KANT, Computer and Information Sciences, Temple University, USA

Misconfiguration is a known and increasingly serious problem in enterprise systems due to frequent code updates and re-tuning of the configuration parameters. Diagnosing complex, residual misconfiguration problems that lead to inaccessible services or failed transactions often starts with either a user complaint or observation by administrators, followed by a largely manual process of deciding what tests to run and how to proceed with further testing based on the test results. The goal of this paper is to automate this process and thereby make root-cause analysis of accessibility related misconfigurations much speedier and much more effective. We explore an extensible domain-knowledge-driven methodology using a network emulator that runs real enterprise networking protocols. Thus, by using commonly used tests, we show that the root-cause can be determined in all cases where discriminative tests exist. The methodology also highlights areas where more discriminative tests are needed to pinpoint the precise configuration variables at fault.

CCS Concepts: • **Networks** → **Network services**; **Network performance evaluation**.

ACM Reference Format:
Negar Mohammadi Koushki, Ibrahim El-Shekeil, and Krishna Kant. 2024. Root-Cause Analysis of Service Misconfigurations in Enterprise Systems. *J. ACM* 000, 000, Article 000 (2024), 30 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Configuration Variables (CVs), or *run-time parameters* are essential for controlling or altering software and hardware system functionalities. It is a common practice to define a large number of CVs for nearly every hardware and software module to provide significant flexibility in satisfying various provider and user needs without coding or other changes. Often, neither the service creators nor the administrators understand how various CVs should be set, their interactions with other CVs, or how the impact of any given CV changes in case of failures or improper setting of other CVs. This lack of clarity is compounded by often absent, outdated, or incomplete documentation of CVs and their interactions.

As a result, misconfigurations have become a prevalent source of operational challenges and are routinely exploited by attackers to gain access or disrupt the system's operation. As the reliance on IT services grows, ensuring their continuous operation and accessibility becomes paramount. Yet, the task of guaranteeing service availability is daunting, given the intricate web of dependencies among services and the sheer number of configuration parameters. The increasing sophistication of these services only adds to the complexity of configurations and dependencies, thereby elevating the risk of misconfigurations and their negative repercussions. Service-Oriented Architecture (SOA) and the emerging

This research was supported by NSF grant CNS-2011252.
Authors' addresses: Negar Mohammadi Koushki, koushki@temple.edu, Computer and Information Sciences, Temple University, 1925 N 12th St, Philadelphia, PA, USA, 19122; Ibrahim El-Shekeil, Computer Science and Cybersecurity, Metro State University, 700 East Seventh Street, Saint Paul, MN, USA, 55106-5000, ibrahim.el-shekeil@metrostate.edu; Krishna Kant, kkant@temple.edu, Computer and Information Sciences, Temple University, 1925 N 12th St, Philadelphia, PA, USA, 19122.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
Manuscript submitted to ACM

microservices paradigm aim to eliminate dependencies between service packages, allowing for faster development, deployment, and easier maintenance and scalability [1]. Continuous Integration/Continuous Development (CI/CD) is the guiding principle of the Development Operations (DevOps) movement, signifying that deployed services are continually improved and fine-tuned, both in terms of code and run-time settings [2]. Furthermore, services are increasingly deployed within lightweight containers, making the hardware resources allocated to the service interchangeable at runtime. Given the constant modifications to service modules, the CI/CD process significantly heightens complexity and the risk of adverse effects. Simultaneously, the growing number of CVs in services not only elevates the chances of malfunctions but also prolongs downtime due to the difficulty in pinpointing the affected component. Consequently, fault diagnosis becomes highly challenging and time-consuming, undermining the quest for high availability.

Misconfigurations can have highly varied impacts on the services, ranging from minor performance issues to situations where many services become inoperational. Misconfigurations in a key infrastructure service like Domain Name Service (DNS) can result in catastrophic impact. For instance, in January 2001, Microsoft's DNS domain suffered inaccessibility due to a misconfiguration where all DNS servers were linked through a single network switch [3]. This setup failed to adhere to the best practices, ultimately causing a complete system breakdown when the switch failed. In September 2009, the entire Sweden domain (`.se`) experienced an hour-long outage due to misconfiguration [4]. In March 2015, Apple's services, including its iOS App Store, iTunes, iBooks, and Mac App Stores, were down for approximately 10 hours due to a DNS error, likely stemming from human error [5].

Additionally, IT systems often grow organically and are configured based on administrator discretion, leading to inconsistency, increased operational complexity, and higher administration costs. This approach also results in unexpected outages when the primary production equipment fails. For example, on August 8th, 2016, Delta experienced a power outage where "some critical systems and network equipment didn't switch over to Delta's backup systems," causing significant disruption [6]. Shortly before Delta's failure, Southwest encountered a failure where "a router failure caused Southwest Airlines' system to crash on Wednesday, July 20, 2016, and all backups failed, causing flight delays and cancellations nationwide and costing the company probably $10 million in lost bookings alone" [7].

In this paper, our focus is exclusively on misconfigurations that make the services inoperational or severely handicapped. That is, our focus is not on misconfigurations that result in softer impacts such as performance degradation or increased security vulnerabilities. We also focus on enterprise *production* systems where intrusive tests (e.g., what-if tests that might change some parameters for testing purposes) or non-standard tests are generally not allowed. Thus, we only use a standard set of tests available to administrators. Our goal is to identify the root-cause of the problem down to the misconfigured CVs, although this may not always be possible and we may need to stop at the level of a larger set of CVs of the affected service that contain the misconfigured CVs. Note that while proactive diagnosis remains an idealized goal, it is usually impractical for large production systems; instead, we assume that the investigation is triggered by a reported problem either by users or administrators (e.g., a failing query or unreachable destination). In such situations, it is not possible to predetermine the tests to be run; instead, the tests are run sequentially based on the insights provided by the previous tests. The goal, thus, is to minimize the amount of testing required to root-cause the problem by intelligently selecting the tests at each step. The key contributions of this paper are as follows:

- We build a comprehensive emulator of enterprise networks using the SEcurity EDucation (SEED) emulator [8], which allows us to run actual open-source code for various services along with their configurations.
- We propose a diagnosis mechanism to root-cause CVs whose misconfiguration impacts the operability of the service. This requires a sequential test selection strategy for which we present a novel algorithm.

- We show that the proposed mechanism works significantly better than the informal approach often used by administrators to guide the testing and is able to automatically root-cause the culprit CVs in all tested cases with less than 20 tests in all cases.
- Our mechanism targets real software implementations used in the enterprise and tests that are widely available and documented. Therefore, our mechanism is not simply a theoretical contribution, but something that can be directly deployed in real enterprise systems.

We believe that our proposed methodology can be easily implemented either in real enterprise networks or their partial "digital twins" constructed using SEED or other emulators. In either case, it will be highly valuable for diagnosing not only the common problems discussed in this paper but also for others with minor changes. We will open-source our code shortly for this purpose, and we believe that would also be a very significant contribution to this work.

The remainder of this paper is structured as follows. Section 2 offers a comprehensive review of the existing literature in network fault diagnosis, highlighting various approaches and methodologies. Section 3 introduces our proposed diagnostic methodology, detailing the foundational concepts and notations used. Section 4 details our approach to diagnosing misconfigurations in interconnected services, discussing factors that influence optimal testing, and providing a detailed step-by-step algorithm for test selection. In Section 5, the design of our network testbed for misconfiguration analysis is presented, which includes the utilization and extension of the SEED emulator. Section 6 presents our experimental results, including the categorization of failures and non-failures and the process of tuning thresholds. The paper concludes in Section 7, where we summarize our findings and discuss directions for future research.

2 RELATED WORK

The concept of utilizing probes to diagnose network faults was initially introduced by Brodie et al. in [9], where they outlined several approximation algorithms for selecting a target probe set for the network. Rish et al. extended this idea in [10, 11], proposing cost-efficient and adaptive diagnostic probing techniques. These techniques employ an information-theoretic approach to choose the target probe set, initially selecting a small set of highly informative probes and dynamically adapting them based on the observed network state, but face weaknesses in computational complexity due to Bayesian network inference. Despite reducing probes by over 60%, real-time suitability is compromised.

Another approach, based on entropy approximation, was presented by Zheng et al. in [12]. This method used a loopy belief propagation model to compute approximate values for marginal and conditional entropy. Natu et al. explored adaptive strategies for probe selection in [13–18], dividing the fault diagnosis process into fault detection and fault localization sub-steps. The fault detection step periodically sends out a target probe set to detect faults, triggering the fault localization step to identify the exact fault location. Lu et al. in [19] proposed an adaptive method inspired by [13], dividing the fault detection process into stages and selecting small probe sets to check network nodes progressively. Tang et al. introduced the Active Integrated Fault Reasoning technique in [20], combining network symptom correlation with active probing for fault localization. This method adapts the probe set based on observed symptoms.

Incremental adaptive probing approaches, suitable for real-time monitoring and diagnosis, select and send probes as needed in response to failures. Carmo et al. in [21] applied active probing for intrusion detection in wireless multi-hop networks using a recursive probe selection scheme and a Bayesian classifier. Garshasbi in [22] proposed an algorithm combining active and passive monitoring for fault diagnosis based on Ant Colony optimizations. These approaches rely on different dependency models for networks, which can be deterministic or non-deterministic. Natu et al. in [13] proposed a probabilistic dependency model representing the relationship between a probe and probed nodes with probability values. However, obtaining accurate probabilistic models with dynamic probabilities is largely infeasible. The

placement of probing stations in the network poses another challenge. Heuristic-based approaches described in [17, 23, 24] incrementally select nodes for probing stations based on the number of independent paths to a node. Traverso et al. in [25] developed a network monitoring tool using a hybrid approach that correlates throughput measurements from probes with alarm notifications from passive measurement utilities. A comprehensive survey of active probing techniques is available in [26].

El-Shekeil et al. in [27] presented the CloudMiner framework for systematic failure diagnosis in enterprise cloud environments. This framework emphasizes probing station selection and utilizes a minimal set of network probes. It is specifically designed to address the complexities and interdependencies of network services and components in cloud systems, aiming to efficiently detect and diagnose failures in these intricate environments.

Integrating root-cause analysis into fault localization is a crucial dimension in advancing the effectiveness of fault identification and resolution. Fault localization, as explored in the previous paragraphs, encompasses various techniques for pinpointing the existence of faults within a network. However, a critical aspect that remains underrepresented is the dedicated focus on understanding the root-causes behind these faults. In the realm of fault management, Lamraoui et al. [28] advocated for a formula-based approach to automatic fault localization in multi-fault programs. Their emphasis on localizing the root-causes underscores the importance of going beyond mere fault detection to delve into the fundamental reasons behind system irregularities. Similarly, Chen et al. [29] introduced the BALANCE method, which employs Bayesian linear attribution specifically for root-cause localization. This multidimensional approach recognizes the need for a nuanced understanding of fault origins, reinforcing the idea that accurate fault localization is incomplete without a corresponding identification of the root-cause.

While the existing literature provides a wealth of fault localization methodologies, Wong et al. [30] offered an insightful overview, emphasizing key faults and complexities in fault localization. Adding a layer of root-cause analysis to these discussions becomes imperative for a holistic understanding of fault management. Zhang et al. [31] contributed by developing a two-level fault diagnosis and root-cause analysis scheme, specifically for interconnected dynamic systems. This work underscores the importance of discerning the root-causes of faults at a granular level, acknowledging that effective fault resolution requires more than just identifying the presence of anomalies.

Our research introduces several novel ideas for effective network fault diagnosis and management, distinguishing it from existing methodologies. Unlike previous approaches that primarily relied on simulations, our work employs the SEED emulator, where the containers run real, unmodified code for various functionalities. Thus, our setup deals with actual networks as they would run in a real data center, with the exception of running them in containers on a single machine rather than on physically distributed hardware. As a result, our testing uses real test commands such as ping, traceroute, nmap, telnet, etc., exactly as they would be in practice. Consequently, our techniques and results apply directly to real systems without the need to develop any special tests, which is always a difficult proposition. This is unlike most studies in the literature where the precise tests either remain unspecified or are described abstractly in terms of nodes that they are applied to or the faults they are sensitive to. Furthermore, we categorize tests into three buckets, namely, name resolution, reachability, and application-level assessments. In each case, we don’t merely check whether a node running various services is up or down; instead, we test their individual CVs to get a detailed understanding of the problem. For example, instead of simply checking if a node is unreachable, we want to understand whether the node IP address, gateway IP address, mask, etc., are misconfigured. Such probing also includes services that maintain repositories relevant to reachability, such as the DNS, DB, firewall rules, etc., that too can be regarded as CVs. Finally, our experimental setup, leveraging the SEED emulator, allows us to replicate a full-scale enterprise network or parts thereof as a kind of “digital twin” that can be used to explore obscure problems in parallel with the actual network continuing to function normally for

transactions that do not trigger the fault. These points collectively underscore the innovative aspects of our work, setting it apart from existing literature in the field.

3 PROPOSED DIAGNOSTIC METHODOLOGY

In this section, we discuss the basics of our diagnosis methodology, starting with a discussion of CVs and faults in enterprise systems.

3.1 Configuration Faults and their Handling

An enterprise system typically runs a large number of services, which includes both the basic system services such as DNS, Dynamic Host Configuration Protocol (DHCP), routing, firewall, etc., and application services such as web-browsing, payment, Database (DB), shopping cart, etc. Each service may involve many CVs of different types. With services such as DNS, routing, firewall, Active Directory, etc. that maintain a repository of specific items (i.e., DNS entries, routing entries, firewall rules, user access privileges, etc.), the configuration includes two types of CVs: (a) *Basic CVs*, that relate to the service as a whole (e.g., IP address of a name server, Virtual Local Area Network (VLAN) setup, etc.), and (b) *Specific CVs*, that relate to individual service entries (e.g., DB entries, DNS entries, routing entries, firewall rules, etc.). The misconfiguration of basic CVs would affect the entire service, whereas the misconfiguration of specific CVs would affect only certain clients or transactions.

Now, suppose that one or more CVs of a service are changed by an administrator either to tune them or due to the CI/CD process that updates the service. This could lead to several scenarios, ranging from benign to insidious. The best-case scenario is the relative insensitivity of the changed CVs to performance or functionality. This happens frequently – many CVs don't do much but were introduced in the name of flexibility without a proper evaluation of their influence. Another case is that the change immediately shows up as misbehavior (e.g., poor performance, unreachability, crash, etc.). However, it may not be desirable to simply revert the change. For example, if we updated a module to handle more network connections because that capability is needed, a simple reversion is a non-solution. Instead, we need to run diagnosis to discover the interaction of this parameter with some existing CVs that led to the misbehavior. Another scenario is that the changes seem to work fine in spite of the bad change, potentially because the effect of the change is triggered only for certain types of transactions or transactional parameters or only under certain conditions such as high CPU utilization, network congestion, or failure of preferred but replicated path or resource. These special conditions may not occur for quite some time, perhaps even days or weeks, and meanwhile, other configuration changes may have occurred to further confuse the fault of what caused the misbehavior. A systematic diagnosis is essential to find the root-cause of the problem(s) in this case. An even more insidious version of this problem is when the impact on CVs is implicit rather explicit. For example, a hardware upgrade, the addition of some device in the system, or the introduction of a new type of query may trigger a previously absent misbehavior.

Configuration problems can be examined both proactively and reactively. A proactive method may run some standard tests following an update or periodically collect detailed logs for offline analysis. While this can be useful, a lack of focus for the tests or log analysis is unlikely to discover many problems. Furthermore, a proactively detected problem would still need to trigger a diagnosis procedure to root-cause the problem. The diagnosis may also be triggered by a user complaint or the administrator noticing some unexpected or unsatisfactory behavior. Although the goal of our diagnosis is finding the root-cause of the misconfiguration, this is not always possible due to the limitations of the generally available tests that we use. In such cases, the goal is simply to narrow down the problem to specific service or group of CVs, which can be investigated further either manually or using more application specific tools.

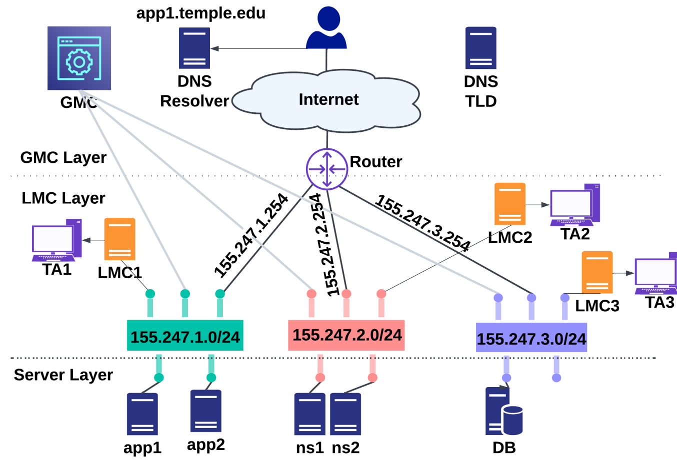


Fig. 1. Our Diagnostic Architecture.

3.2 Diagnosis Architecture

The complexity of diagnosing problems in IT systems arises from the intricate interdependencies among services and their CVs. Typically, the diagnostic process begins with a user-reported fault or an observation by an administrator, which may manifest as application slowdowns, query failures, or resource inaccessibility. Our diagnostic framework employs the concepts of Local Management Consoles (LMCs), Test Agents (TAs), and a Global Management Console (GMC), as illustrated in Fig. 1. To effectively narrow down the scope of the problem, we operate under the assumption that the underlying fault is permanent, although it might only become apparent under specific conditions such as heavy loads, resource pool exhaustion, or certain types of queries.

We further assume that the network is composed of multiple subnets, potentially including VLANs. Each subnet can be thought of as a local cluster comprising a number of servers and/or clients. Within each subnet, an LMC provides real-time updates on service statuses and serves as the initial point of fault detection. Each LMC is a crucial component of the local management network for its respective subnet and functions independently of the main network's state. This setup of having an independent local management network is commonplace in enterprise environments and reflects real-world scenarios accurately. For simplicity, we assume that these management networks are highly robust and are not the focus of our diagnostic procedures.

TAs can be thought of as software programs running in each subnet and are responsible for initiating local tests and displaying results via the LMCs. TAs are also used to initiate non-local tests for diagnosis purposes and may additionally collect data, analyze it, and log it for future examination.

A critical aspect in diagnosing enterprise systems is the limited visibility of service status beyond local clusters or points of presence, which we emulate using subnets. For example, if a client c in $subnet_1$ is unable to access service s in $subnet_2$, the root-cause, whether it's a network fault or service s being offline, remains unclear. Resolving this typically requires attempting access from another subnet. However, failure from this secondary attempt doesn't conclusively indicate the operational status of service s, although repeated failures from different subnets bolster the likelihood of it being offline.

To mitigate this, we posit that the service status list from each LMC should be accessible across subnets, allowing any client to ascertain the operational status of services network-wide. This setup effectively assumes the existence of a global

console, GMC, acting as a centralized entity where results from various LMCs can be consolidated. In this paper, we assume the existence of such a GMC, which is always available for consultation without excessive delay. There are several issues in supporting such an abstraction, but dealing with them is beyond the scope of this paper and will be addressed in future work. Briefly, the GMC abstraction can be achieved by defining a redundant, multi-path overlay network so that all LMC-to-LMC communications can occur robustly and without excessive delays. However, one must still deal with the distributed systems issues here, such as the remotely observed LMC state being somewhat stale. We reserve consideration of such issues for future work.

3.3 Testing Framework Basics

Consider an enterprise system with n services denoted as $S = \{s_1, s_2, \ldots, s_n\}$. Each service s_i involves m_i CVs denoted by the set $CV_i = \{cv_{i1}, \ldots, cv_{im}\}$. We also define a severity measure π_i for the impact or criticality of service s_i. This severity index is essential for prioritizing services for testing. We assume $\pi_i \in [0..1]$ where 0 indicates no impact or low criticality, and 1 signifies maximum impact or high criticality. The assignment of severity values is based on domain knowledge, historical data, and the perceived consequences of service failures. We also define a permutation P of the service indices that arranges them in the decreasing order of severity, i.e., $\pi_{P(i)} \geq \pi_{P(j)}$ for all $j > i$.

Let $T = \{t_1, t_2, \ldots, t_l\}$ denote the complete set of tests, and let us define three functions to connect them to services and their CVs. Let $T_R(s_i)$ be the set of tests relevant to service s_i, and $CV_{t_{s_i}}$ be the set of CVs involved in the test t for service s_i. A test t is considered relevant to service s_i if and only if it involves one or more of the CVs associated with s_i (i.e., $CV_t \cap CV_i \neq \emptyset$). In other words, a test is relevant to a service if it has the potential to detect issues or validate the functionality of the CVs within that service. We also assume that each test either passes or fails. We denote this using the function Outcome(t) with values 1 (pass) and 0 (fail).

3.3.1 Defining Diagnostic Tests and Their Categories.

A test in our system validates specific aspects of functionality either explicitly or implicitly through the returned response. For instance, a test ensuring communication with a remote DB implies operational network connection, routing, DNS entry, firewall rules, DB connector service, and the DB itself. However, it doesn't confirm that the network or the service is configured properly from a performance perspective. Additional tests focused on examining the CVs of various services are needed to establish the proper configuration of the CVs. In Fig. 2, a test has input CVs ($\{cv_{1i}, cv_{2i}, ..., cv_{ji}\}$), a test command ($cmd_e$), and processed/filtered CVs ($\{cv_{1i}, cv_{2i}, ..., cv_{ki}\}$) as output. Outcome$(t)$ helps determine if CVs in the test pass or fail.

A test type comprises of tests using the same command but different inputs from various services. For example, in Table 1, using the same CVs with different values from services s_i and s_j for the "nmap" command assesses connectivity and accessibility of a given service, respectively.

Table 1. Test and Test Type Definition.

Test Type	Test	Input	Command	Output
Test Type1	t_1	app1.temple.edu	nmap	10.10.1.1, $tcp/80$
	t_2	app2.temple.edu		10.10.2.1, $udp/53$

Testing of remote functionality involves three distinct steps: name resolution of the service, accessibility via the network, and service functionality itself. This motivates us to define the following hierarchy (or sequence) for testing.

Name Resolution (DNS queries): Placed first as the foundational layer of network communication, it establishes a stable reference point by addressing the reliability and consistency of DNS name resolution.Key assumptions include the

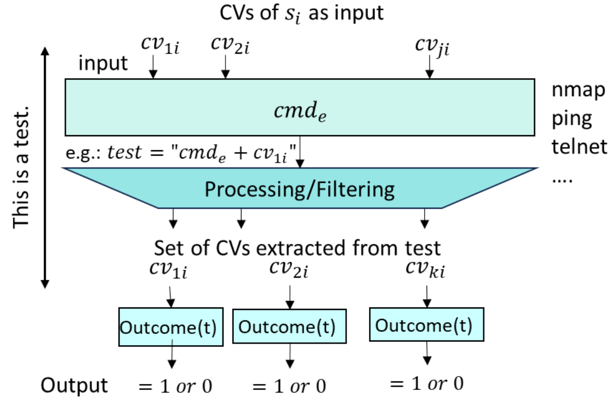


Fig. 2. Test Definition.

uniformity of domain names across clients, DNS resolvers consistently reaching authoritative root servers, consideration of client and resolver variations, and simplified assessment through domain administrator configurations.

Reachability (ping, traceroute, etc.): After name resolution, we need to evaluate the network's ability to reach destinations, considering obstacles like firewalls and routing faults. Assumptions involve both general and specific reachability, the nuanced interpretation of failed pings, the necessity of additional tests like traceroute, and the implication of a successful ping indicating necessary route information.

Application (telnet, HTTP get, etc.): The final category examines specific network services and applications, logically following name resolution and reachability confirmation. Assumptions encompass generic tests like telnet for Transmission Control Protocol (TCP) services, the significance of responsive applications indicating operational services, non-responsive applications indicating potential faults, and the disruptive role of firewalls in application functionality.

3.3.2 Services vs. Faults. When the LMC indicates that certain services are deemed to be faulty, we need an indication of potential faults that may be affecting those services. These faults may include things like isolated service malfunctions, server failures, network and routing faults, DNS and firewall problems, service invocation, accessibility from the client, and configuration settings in files. Subsequently, we assess which potential faults could simultaneously impact all problematic services.

We assume that the connection between services and potential faults is determined from the domain knowledge and is looked up for diagnosis. We note that a single fault could potentially affect more than one service. For example, suppose that the LMC indicates that both services s_1 (App server) and s_2 (DNS server) are faulty. In addition to both services truly being faulty, it is also possible that the observed problem in s_1 is caused by a true problem in s_2 or vice versa. This simultaneous fault manifestation could be attributed to a shared network issue affecting the communication between the app server and the DNS server, leading to disrupted service functionality. Furthermore, misconfigurations in the DNS settings might impact both the app server and the DNS server, causing a cascading effect on their performance. For example, if there is a misconfiguration leading to a discrepancy in IP address resolution, it could affect the communication between services, leading to service degradation. For the diagnosis we will first determine all the faults that could affect these services and find the tests relevant for those faults.

3.3.3 Weight of the Test. Different tests have different levels of effectiveness. In much of the work in the literature, this aspect is considered only in terms of how many "nodes" a test involves. In reality, the fault of effectiveness is far more

complex and requires a consideration of the semantics of the test that we try to capture using a couple of different concepts, the most important one being the *weight of a test* relative to a fault. It is difficult to assign the weight automatically; however, it is easy to tell from the domain knowledge how useful a given test is for specific faults. To exploit such domain knowledge in testing, we start with a simple matrix of faults vs. tests, where the matrix entry is assigned manually one of the three numbers: 0: test has no relevance to the fault, 1: test is highly relevant for the fault, and 0.5: Test is either partly relevant or it is difficult to determine the relevance (e.g., relevant in certain situations). This matrix is built only once and will be largely static except for an occasional update. The matrix is not obligated to list all possible faults but only the known and significant ones. Its purpose is to obtain the overall weight of a test (by summing the assigned weights for all the faults considered). Note that since we do not know the fault during diagnosis, only an overall test weight is usable in the algorithm. The weights can be normalized to 0..1 range for uniformity. We henceforth denote the weight of a test t as $\omega(t)$.

For example, the test '`host <DNS record>`' (refer to Table 2) is considered highly relevant for 'DNS Entry' due to its direct association with resolving domain names. However, it is assigned no relevance for 'Firewall Entry', as its primary function is not firewall-specific. Conversely, '`nmap <ip addr>`' is deemed highly relevant for both 'Firewall Entry' and 'Network Connection and Routing' because it can identify open ports and assess network connectivity. Also, consider the test '`ip` -4 $addr$ $show$ dev $< interface >$' with a relevance score of 0.5 for the 'Network Connection and Routing' fault. It offers insights into IPv4 addresses on a specific interface, but its direct contribution to overall network and routing issues is not definitive.

3.3.4 The Concept of Confidence Level for Configuration Variables.

Since our tests are focused on root-causing the reported problem down to the specific CVs of the services, we need a measure of which CVs are likely to be misconfigured and which are likely to be fine. We do this by associating a *Confidence Level* (CL) with each CV in the range [-1,+1], where -1 means that we strongly believe that the CV is set to a value outside of its acceptable or sensible range, and +1 means the opposite (i.e., we believe that the CV is set properly and not contributing to the observed problem). The middle value of 0 indicates that we have no information about its relevance to the fault. At the start of the diagnosis, for every CV m of every service i, we initialize its CL, denoted $Q(cv_{im})$ either to 0 (if there is no prior knowledge about the relevance of this CV to the observed problem) or to some small positive/negative value (if there is prior knowledge).

The next issue is how much we change $Q(cv_{im})$ following a test, henceforth denoted as $\Delta Q_{im}(t)$. We estimate it using the weight of the test and the number of CVs involved in the test. That is, $\Delta Q_{im}(t) = \frac{\omega(t)}{|CV_t|}$, where $\omega(t)$ is the test's weight and $|CV_t|$ is the number of CVs involved in the test t. If Outcome$(t) = 1$ (test passes), all CVs in CV_i have their confidence increased, i.e., $Q(cv_{im}) += \Delta Q_{im}(t)$. If Outcome$(t) = 0$ (test fails), all CVs in CV_i have their confidence decreased, i.e., $Q(cv_{im}) -= \Delta Q_{im}(t)$.

4 PROPOSED APPROACH FOR FAULT DIAGNOSIS OF MISCONFIGURATIONS

This section outlines our framework for diagnosing service misconfigurations in interconnected enterprise IT systems. Our approach utilizes a systematic domain-knowledge driven algorithm for test selection. The algorithm aims to satisfy two competing goals: minimize the number of tests while maximizing the likelihood of accurately uncovering misconfigurations. In the following we will introduced some more notations and terms, all of which have been collected into Table 3 for ease of reference.

Table 2. Sample Matrix of Test Relevance for App Server Fault Diagnosis.

Test \Fault	DNS Entry	Firewall Entry	Network Connection & Routing
host $< DNS\ record >$	1	0	0.5
ping $-c5\ < ip\ addr >$	0.5	0.5	1
traceroute $< ip\ addr >$	0.5	0.5	1
nmap $< ip\ addr >$	0	1	1
ip $-4\ addr\ show\ dev\ < interface >$	0	0	0.5
ifconfig $< interface >$	0	0	0.5
iptables $-Lv$	0	1	0.5
telnet $< ip\ addr >\ < port >$	0.5	0.5	1

Table 3. Summary of Notations and Abbreviations.

Notation/Term	Definition	Notation/Term	Definition
n	Number of services	M	Matrix of faults vs. tests
S	Set of services	F	Set of potential faults
m_i	Number of CVs for s_i	$Q(cv_{im})$	CL associated with cv_{im}
CV_i	Set of CVs for s_i	$\Delta Q_{im}(t)$	Change in CL of cv_{im} following t
π_i	Severity measure for s_i	D_{ij}	Dependency matrix
P	Descending permutation	MT	Misconfiguration threshold
T	Complete set of tests	ZC	Threshold of CVs with zero CL
$T_R(s_i)$	Tests relevant for s_i	WT	Weight of the test threshold
CV_m	Set of misconfigured CVs	$\mathcal{T}_{\text{selected}}$	Set of selected tests
$CV_{t_{s_i}}$	CVs in test t for s_i	AS	Autonomous System
Outcome(t)	Outcome of test t	BGP	Border Gateway Protocol
$\omega(t)$	Weight of test t	CL	Confidence Level
$W(t)$	Normalized weight of t	DB	Database
$Z(t)$	Count of CVs with CL zero in t	DNS	Domain Name System
$\mathcal{T}_{\text{selected}}$	Selected tests	GMC	Global Management Console
t_{mc}	Test with max zero CLs	LMC	Local Management Console
t_{mw}	Test with max weight	SEED	Security Education
t_{optimal}	Optimal test selected based on criteria	TA	Test Agent
s_{picked}	Service associated with t_{optimal}	TCP	Transmission Control Protocol
CV_{picked}	Set of CVs associated with s_{picked}	VLAN	Virtual Local Area Network
S_p	Problematic services shown in LMC	VRRP	Virtual Router Redundancy Protocol

When a user complaint is received, the fault may lie either with some services in some subnet or with the network. In the former case, the corresponding LMC will show the faulty services, of which there could be several because of potential dependencies between services. In this case, we need to choose tests that are relevant to the faulty services. Otherwise, we start a systematic diagnosis procedure that checks for network related issues as discussed in section 3.3.1.

4.1 Overall Diagnosis Algorithm

In this section, we introduce an algorithm designed for diagnosing potential misconfigurations across multiple services. The algorithm takes a list of problematic services shown in LMC, denoted as $S_p = \{s_1, \ldots, s_j\}$, and a Misconfiguration Threshold (MT) as input. In the initialization phase, the algorithm establishes sets $\mathcal{T}_R$, $\mathcal{CV}_t$, and $\mathcal{CV}$. Specifically, it includes all relevant tests associated with the specified services in $\mathcal{T}_R$, set of CVs linked to these tests for these services in $\mathcal{CV}_t$, and CVs associated with the specified services in $\mathcal{CV}$ (Lines 2-6). The overall algorithm (Algorithm 1) iteratively executes a sequence of steps until either the absolute value of CLs of all CVs are above the MT or a misconfiguration is identified.

Algorithm 1: Diagnose Faults

1 **Function** DIAGNOSEFAULTS(S_p, MT)
2 Initialize $\mathcal{T}_R$, $\mathcal{CV}_t$, $\mathcal{CV}$;
3 **for** *each* $s_i \in S_p$ **do**
4     $\mathcal{T}_R = \mathcal{T}_R \cup T_R(s_i)$;
5     $\mathcal{CV}_t = \mathcal{CV}_t \cup CV_{t_{s_i}}$;
6     $\mathcal{CV} = \mathcal{CV} \cup CV_i$;
7 **while** $\forall Q(cv) > MT, cv \in \mathcal{CV}$ **do**
8   Call `TestSelectionExecution`($\mathcal{T}_R$, $\mathcal{CV}_t$, $\mathcal{CV}$);
9   **for** *each* $s_i \in S_p$ **do**
10   **if** $\nexists t \in T_R(s_i)$ ***and*** $\nexists Q(cv) < MT$ **then**
11             $cv^{(s_i)}_{\text{misconfig}} = \text{argmin}_{cv \in \bigcup_{t \in T_R(s_i)} CV_{t_{s_i}}} (\min(Q(cv)))$;
12 **return** Set of **misconfigurations** CV_m;

Algorithm 2: Test Selection and Execution

1 **Function** TESTSELECTIONEXECUTION($\mathcal{T}_R$, $\mathcal{CV}_t$, $\mathcal{CV}$)
2 $Z(t) = \sum_{cv \in \mathcal{CV}_t} \delta(Q(cv) = 0), \quad \forall t \in \mathcal{T}_R$;
3 **if** $\exists t\ Z(t)$ **then**
4     $\mathcal{T}_{\text{selected}} = \{t \mid \omega(t) \geq WT$ **and** $Z(t) \geq ZC\}$;
5   **if** $\mathcal{T}_{selected} \neq \emptyset$ **then**
6         $t_{\text{optimal}} = \text{argmax}_{t \in \mathcal{T}_{\text{selected}}} \omega(t)$;
7   **else**
8         $t_{\text{optimal}} = \text{argmax}_{t \in \mathcal{T}_R} Z(t)$;
9 **else**
10     $\mathcal{T}_{\text{selected}} = \{t \mid \min_{cv_j \in \mathcal{CV}_t} Q(cv_j) = \min_{t_y \in \mathcal{T}_R} \min_{cv_l \in CV_{t_y}} Q(cv_l), t \in \mathcal{T}_R\}$;
11     $t_{\text{optimal}} = \text{argmax}_{t \in \mathcal{T}_{\text{selected}}} |CV_t|$;
12   s_{picked} = Service associated with t_{optimal};
13 Outcome(t_{optimal}) = Execute(t_{optimal});
14 Call `UpdateConfidenceLevels`(s_{picked}, CV_{picked}, t_{optimal});

Within each iteration, it calls Algorithm 2 to strategically select and execute tests (Line 7-8). Following the test execution, the algorithm checks for the presence of relevant tests and misconfigurations for each service. If all relevant tests for the service run but no misconfigurations are detected, the algorithm identifies a potential misconfiguration based on the most negative CL for that service. The final output is a set of identified misconfigurations, each corresponding to a service along with the misconfigured CV (Lines 9-12).

The algorithm presented in Algorithm 2 is a crucial component of the overall diagnostic process and is responsible for the strategic selection and execution of optimal tests based on the outcomes of previous diagnostic steps. Let's break down the key steps and functionalities of this algorithm:

For each test t in the set of relevant tests $\mathcal{T}_R$, the algorithm computes $Z(t)$, the number of CVs with zero CL value. It also checks if the confidence level $Q(cv)$ of any CV associated with the test is equal to zero (Line 2).

If there exists a test with at least one CV having a confidence level of zero (Line 3), the algorithm identifies a subset of tests, $\mathcal{T}_{\text{selected}}$, that meet conditions involving WT and ZC (Line 4). If $\mathcal{T}$selected is not empty, the optimal test ($t_{optimal}$) is

Algorithm 3: Update Confidence Levels

```
1 Function UPDATECONFIDENCELEVELS(s_picked, CV_picked, t_optimal)
2 for each cv_m ∈ CV_picked do
3   if Outcome(t_optimal) = 1 then
4     Q(cv_picked,m) = Q(cv_picked,m) + ΔQ_picked,m(t_optimal);
5   else
6     Q(cv_picked,m) = Q(cv_picked,m) − ΔQ_picked,m(t_optimal);
7   if Q(cv_picked,m) < MT then
8     return cv_picked,m as misconfiguration for service s_picked;
```

set to the test with the maximum weight among those selected (Lines 5-6), prioritizing high-weight tests such as ping and nmap. If $\mathcal{T}_{\text{selected}}$ is empty, the optimal test is set to the test with the maximum count of CVs having zero CLs (Lines 7-8).

If no test has zero-confidence level CVs, the algorithm identifies a subset of tests, $\mathcal{T}_{\text{selected}}$, based on minimum CLs across CVs associated with those tests (Lines 9-10). The optimal test ($t_{optimal}$) is selected from $\mathcal{T}_{\text{selected}}$ based on the count of CVs associated with the test to explore more knowledge about more CVs by conducting that test (Line 11). s_{picked} in line 12 indicates the service associated with the optimal test. Finally, the chosen optimal test ($t_{optimal}$) is executed, and the outcome (pass or fail) is recorded (Line 13). The algorithm then proceeds to call Algorithm 3 to update the CLs of CVs based on the outcome of the executed test.

Algorithm 3 takes three key parameters: s_{picked}, representing the service associated with the optimal test, CV_{picked}, representing the set of CVs associated with the picked service, and $t_{optimal}$, which is the test that was deemed optimal and executed.

The core of the algorithm involves iterating through each cv_m in CV_{picked}. Depending on the outcome of the optimal test ($t_{optimal}$), the CL $Q(cv_{picked,m})$ of each CV is adjusted. If the test succeeds, the CL is increased by a certain amount $\Delta Q_{picked,m}(t_{optimal})$. Conversely, if the test fails, the CL is decreased by the same amount (Lines 2-6). Following the CL updates, the algorithm checks if any CV's CL has fallen below the predefined misconfiguration threshold (MT). If such a condition is met, the algorithm promptly identifies the CV as a potential misconfiguration for that service (Lines 7-8).

The proposed algorithms aim to minimize the number of tests while effectively diagnosing misconfigurations across multiple services by intelligently selecting tests based on their weight, the count of variables with zero CLs, and the outcome of previous tests. The thresholds (MT, WT, ZC) are tuned in Section 6.3 to achieve a balance between test efficiency and accurate misconfiguration detection.

4.2 Illustration of Establishing Sets $\mathcal{T}_R$, $\mathcal{CV}$, and $\mathcal{CV}_t$ in the Initialization Phase

Fig. 3 illustrates two services, s_1 and s_2 which the LMC indicates as being faulty. As stated earlier, in addition to both services truly being faulty, it is also possible that the observed problem in s_1 is caused by a true problem in s_2 or vice versa. Thus, we start by identifying a set of potential faults denoted as $F = \{f_1, f_2, \ldots\}$ that can affect these services. We do this on the basis of a repository constructed using the domain knowledge. In the figure, four faults ($f_1, f_2, f_3,$ and f_4) are identified, and red dashed lines point to the boxes of the services that they can affect. It is seen that f_1 and f_2 can affect both s_1 and s_2, whereas f_3 and f_4 impact only one of the problematic services.

In the top right corner of the figure, a set of tests (t_1, t_2, t_3, t_4, t_5) is presented. As explained in Section 3.3.3, we figured out how useful each test is for specific problems, shown by black hashed lines. In this figure, $t_1, t_2,$ and t_3, comprising the

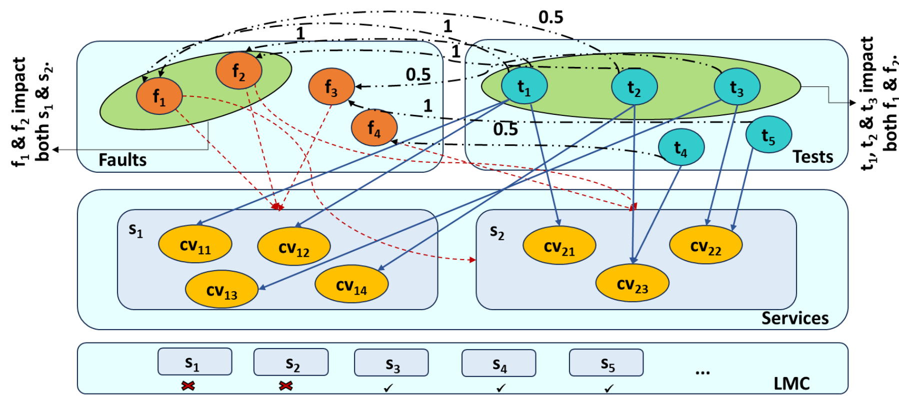


Fig. 3. Illustration of Test Selection When Two services are Faulty.

$\mathcal{T}_R$ set, are identified as tests that can impact the potential faults, f_1 and f_2. The focus now shifts to evaluating whether these tests can help in diagnosing CVs of problematic services.

In the middle of the figure, the problematic services and their relevant CVs are displayed. CVs for each service ($\mathcal{CV}$) are presented alongside the impact of each test on these CVs, depicted by blue lines, which establish the $\mathcal{CV}_t$ set.

5 DESIGNING THE NETWORK TESTBED FOR MISCONFIGURATION ANALYSIS

In studying misconfiguration diagnosis, it is essential to set up a reasonably sized enterprise network that supports not only the basic packet routing but also all important services, including those that are network-related (e.g., Firewall, DNS, Network Address Translation (NAT), etc.) and those that support client queries (e.g., web-service, DB service, File Transfer Protocol (FTP), etc.) Simulation is not a viable option for such a setup because of the lack of comprehensive simulators; thus, the only options are testbeds and emulators. Several testbeds are available with different strengths as surveyed in [32] – the better-known ones being EmuLab and GENI. Although either of these could, in principle, be used, setting up a complete environment using these testbeds takes quite a bit of effort. Network emulators provide a similar capability by providing the ability to spin up Virtual Machines (VMs) or containers on a local machine to implement various functionalities. For example, open-source routing implementations such as Bird can be run in the VMs/containers to emulate routing protocols, BIND for DNS service, MariaDB for DB service, Apache for Web service, etc. Such a network will have all of the same capabilities, commands, and configuration parameters as a real implementation, and thus the diagnosis techniques can be used unaltered in real networks. Furthermore, unlike testbeds, it is possible to introduce additional network delays easily.

A well-known network emulator is CORE (Common Open Research Emulator) which we have experimented with extensively in the past [33]. It provides the capability to run popular data-center network routing protocols such as Open Shortest Path First (OSPF) and can support nodes running arbitrary code, which are set up as lightweight VMs. Unfortunately, we were unable to automate the job of running commands inside the VMs and returning results back to our control node, which is essential to conveniently manage a large network. Instead, we used a recently developed emulator called SEED, which is an open-source Python library designed to emulate the Internet for educational purposes [34]. In the following, we provide a brief description of SEED, its limitations, and the extensions that we made.

5.1 Utilizing and Extending SEED for Network Misconfiguration Studies

SEED Emulator, with its comprehensive Python classes, mirrors key components of the Internet to emulate (autonomous systems) or ASes, networks, hosts, and Border Gateway Protocol (BGP) routers, along with services like Web servers,

DNS, and various cybersecurity scenarios (e.g., Botnets, Darknets), enables the construction of a mini-Internet for realistic emulation. These emulations, encapsulated within Docker containers, facilitate diverse cybersecurity and networking experiments. The extensibility of SEED is a notable feature, allowing for the development of new classes to simulate complex configurations, such as an Ethereum blockchain.

However, its emulation capabilities do encounter certain limitations, especially when replicating specific aspects of enterprise networking, such as Layer 2 (L2) switching and high availability for gateways.

A notable area where SEED's default setup diverges from typical enterprise network configurations is in its approach to network interconnections. SEED primarily focuses on connections via Internet Exchanges (IXes). This model, while comprehensive, does not entirely reflect the direct Internet Service Provider (ISP) peering arrangements often employed in enterprise networks, where different locations, including headquarters, data centers, and branch offices, commonly establish direct peering with various ISPs for improved connectivity and redundancy.

Nevertheless, within SEED's repository, particularly in the not-ready-examples section, we discovered an example that lays the groundwork for direct ISP peering. We leveraged this example, refining and adapting it to more accurately emulate the type of connectivity observed in enterprise networks. This adaptation has been instrumental in bridging the gap between SEED's standard emulation capabilities and the specific needs of enterprise network scenarios.

In addition to addressing the interconnection aspect, we extended SEED's functionalities to include the Virtual Router Redundancy Protocol (VRRP).

VRRP is essential for high availability and fault tolerance in enterprise networks. It enables seamless gateway failover by creating a virtual IP address shared among multiple physical routers. In the figure (Fig. 4), this is depicted with a host and two routers connected to an Ethernet switch. The host's default gateway is the virtual IP address 10.0.0.1, while the physical routers have addresses 10.0.0.2 and 10.0.0.3. These routers communicate to determine a master router, which is responsible for forwarding traffic to the virtual IP address. If the master router fails, a backup router automatically assumes the master role, maintaining uninterrupted network service.

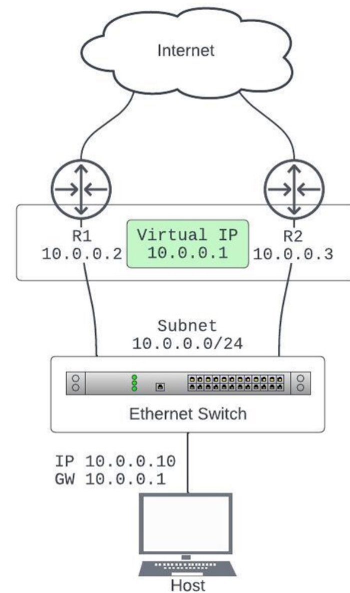


Fig. 4. Illustration of VRRP.

These enhancements, especially the incorporation of VRRP and the adaptation of direct ISP peering configurations, have notably advanced SEED's utility for studying network misconfigurations in environments that mirror actual enterprise settings. By doing so, we have expanded the potential of SEED as a platform for developing diagnostic algorithms and tools tailored to the intricacies of modern network infrastructures. We illustrate this by emulating two prototypical networks in SEED as described next.

5.2 Case Study: Diagnosing Misconfigurations in a Compact Network Setup

This is a compact network infrastructure and includes a router, an internet connection, and multiple servers. Owned by a single entity, this setup has unrestricted visibility and testing across all resources. For clarity, our focus is on an Ethernet switch with multiple VLANs. VLANs typically segment workloads to minimize broadcast domains and enhance security. If a server within a VLAN is compromised, its impact remains contained. A router facilitates inter-VLAN routing. In our

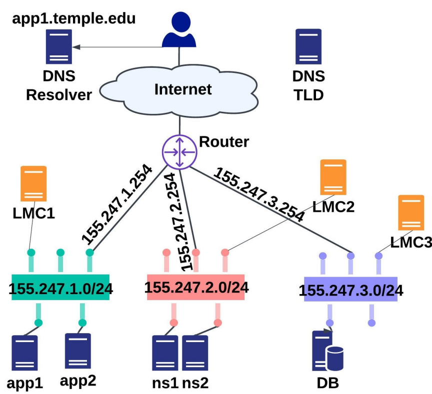


Fig. 5. Compact Network.

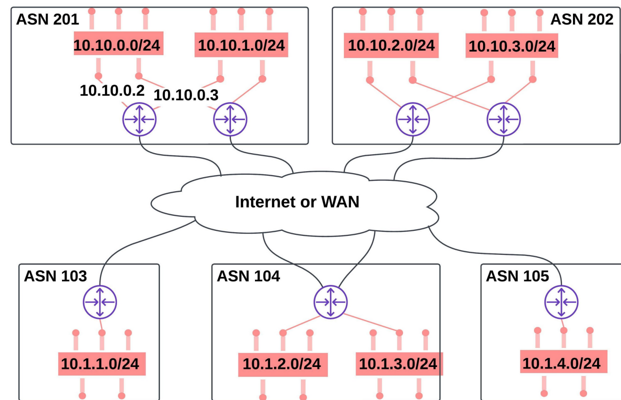


Fig. 6. Complex Network setups.

Table 4. IP Configuration of Network Devices with Misconfigurations Highlighted.

Host	IP Address	Subnet mask	Gateway
App1	155.247.1.1	255.255.0.0	155.247.1.254
App2	155.247.1.2	255.255.255.0	155.247.1.254
ns1	155.247.2.1	255.255.255.0	155.247.2.254
ns2	155.247.2.2	255.255.255.0	155.247.2.254
DB	155.247.3.1	255.255.0.0	155.247.3.254
LMC1	155.247.1.10	255.255.255.0	155.247.1.254
LMC2	155.247.2.10	255.255.255.0	155.247.2.254
LMC3	155.247.3.10	255.255.255.0	155.247.3.254

Table 5. Diagnostic Test Results for the App1 Access Misconfiguration Scenario.

Source	Destination	Result
LMC1	App1	Fail
LMC1	App2	Pass
LMC1	ns1	Pass
LMC1	ns2	Pass
LMC1	DB	Pass
LMC3	DB	Pass

Table 6. Diagnostic Test Results for the DB Misconfiguration Scenario.

Source	Destination	Result
LMC1	App1	Fail
LMC1	App2	Fail
LMC1	ns1	Pass
LMC1	ns2	Pass
LMC1	DB	Fail
LMC3	DB	Pass

example, we assume the router solely provides routing services and public IP addresses are in use, eliminating the need for network address translation (NAT). This network is depicted in Fig. 5.

Within this network, we have two application servers, a DB server, two name servers, three VLANs, and an LMC/TA for each VLAN. External users connect to the internet and use a public DNS resolver. All servers operate under the “temple.edu" domain. When users query the DNS resolver for server names, the DNS Top Level Domain (TLD) server directs them to the network’s name servers: ns1.temple.edu and ns2.temple.edu. These servers are the authoritative name servers for the “temple.edu” domain.

Our network includes TAs that serve as regular users and diagnostic tools capable of normal network access and specialized diagnostic tasks. The network hosts two main application servers, App1 and App2, as shown in Fig. 5. TA LMC1 is connected with App1 and App2 on the same Ethernet switch in VLAN A. Name servers ns1 and ns2, along with TA LMC2, are in VLAN B, managing domain name resolutions. The DB server and TA LMC3 are located in VLAN C.

Each device's IP configuration is detailed in Table 4. App1 and App2's functionality critically relies on the DB server's availability. However, due to the limitations of the SEED emulator, each VLAN is represented as a separate network in our configuration.

Table 4 presents the IP configurations for the network depicted in Fig. 5. In this setup, two misconfigurations are notable: App1's subnet mask is incorrectly set to 255.255.0.0, and the DB server's subnet mask is also misconfigured as 255.255.0.0. This configuration causes the DB server to mistakenly assume that App1 and App2, with IP addresses 155.247.1.1/16 and 155.247.1.2/24, respectively, are within its local subnet (155.247.3.1/16). As a result, the DB server incorrectly routes responses meant for App1 and App2 to its local network instead of the intended router, leading to communication failures. These misconfigurations and their subsequent network faults are further elucidated in Tables 6 and 5, which display the diagnostic results of these faults.

5.3 Exploring Advanced Misconfiguration Scenarios in Complex Network Environments

In large enterprise environments, the intricacy of network configurations significantly heightens the risk and impact of misconfigurations. Unlike more straightforward setups, these complex networks often involve multiple layers of firewalls, routers, and redundancy protocols like VRRP, each adding numerous CVs. This multitude of CVs not only increases the potential for misconfiguration but also makes the fault diagnosis more challenging.

One common problem in such environments is asymmetric routing, particularly in locations with multiple gateways. Traffic might exit the network via one gateway but return through another. Inline firewalls, which expect symmetric traffic flow, may block this returning traffic, leading to disruptions often seen in routing and firewall misconfigurations.

Additionally, implementing high-availability configurations like VRRP, while beneficial for network resilience, further complicates the landscape. These setups introduce more CVs, increasing the chances of misconfigurations and making diagnosis more intricate. For instance, our network model, as shown in Fig. 6, simulates a typical enterprise core network. It comprises interconnected routers and diverse location setups, each varying in complexity from single-router connections to dual-router configurations with VRRP. In such network structures, the interplay between various network elements — from routers to firewalls — and their respective configurations necessitates a comprehensive analysis for effective troubleshooting.

In the next section, we present experimental results for the diagnosis of a variety of faults injected into both the compact and complex networks introduced here.

6 EXPERIMENTAL RESULTS

To comprehensively validate the robustness of our approach, we aim to explore various scenarios related to the LMC feedback. Firstly, we intend to examine cases where the LMC indicates service failures, such as when one service s_i, or multiple services $\{s_1, ..., s_j\}$ are reported to be experiencing faults. This monitoring approach allows for timely identification and prioritization of problematic services. Additionally, we recognize the importance of investigating situations where the LMC does not report any failures, but clients within a subnet raise complaints regarding service faults. In such instances, we delve deeper into understanding discrepancies between client-reported problems and the LMC's status assessments. This dual-pronged investigation strategy will provide a comprehensive perspective on the efficacy of our network monitoring and management systems, ensuring a reactive response to both LMC-identified failures and client-observed faults.

Table 7. Elapsed Time (sec) for Finding Misconfigured IP address of APP server.

Number of tests	Test	Time
—	Preparation	0.46
t_1	host < $DNS\ record$ >	2.49
—	Preparation	0.34
t_2	nmap < $ip\ addr$ >	1.51
—	Preparation	0.27
t_3	ping $-c$ 5 < $ip\ addr$ >	14.27
Total	—	**19.34**

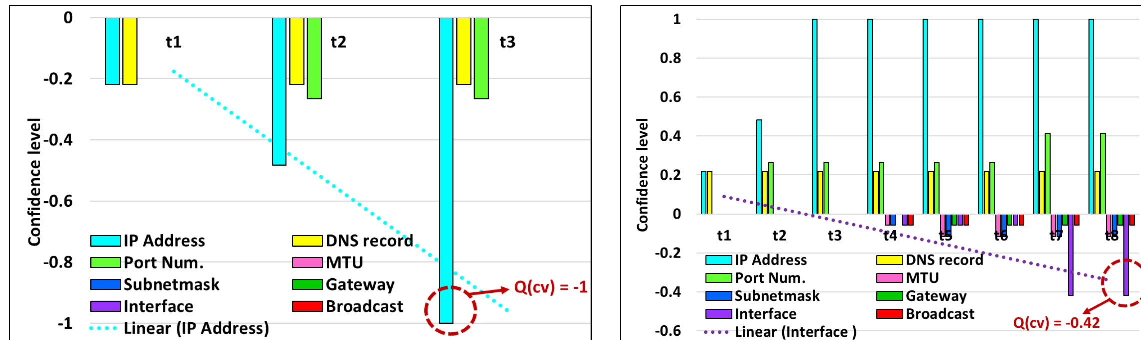
Table 8. Assessing CL of CVs for finding Misconfigured IP address of APP server in each test.

CVs	t_1	t_2	t_3
IP Address	-0.22	-0.48	**-1**
DNS record	-0.22	-0.22	-0.22
Port Num.	0	-0.26	-0.26
MTU	0	0	0
Subnet mask	0	0	0
Gateway	0	0	0
Interface	0	0	0
Broadcast	0	0	0

6.1 LMC Shows Failures

In this section, we will assess the impact of misconfigurations in APP servers, DB servers, and DNS servers by deliberately introducing various misconfigurations that LMC is able to report. This analysis spans from small local clusters to larger subnets and considers strategies for detecting and resolving faults across diverse network sizes and server setups.

6.1.1 Misconfigured IP Address for App Server. Here we focus on the complex network size, which consists of 17 servers distributed across 5 AS'es. The algorithm is executed in several stages, including test selection, DNS record verification, IP address inspection, nmap scanning, and connectivity checks. The entire procedure takes 19.34 seconds, detailed in Table 7. Notably, after the first 3 tests, the algorithm detects a misconfigured IP address with a CL of -1, indicating a strong likelihood of service failure due to misconfiguration (see Fig. 7a and Table 8). The figure illustrates how CLs for various CVs change with each test. For instance, after the first test (t_1 : $host$ < $DNSrecord$ >), the CL for IP address and DNS record decreases to -0.2. Subsequently, after t_2 : $nmap$ < $ipaddr$ >, it further drops by -0.26. Finally, during t_3 : $ping - c5$ < $ipaddr$ >, the CL for the IP address hits -1, leading the algorithm to stop as it identifies the misconfiguration.



(a) Misconfigured IP Address on the App Server. (b) Misconfigured Interface on the DB Server.

Fig. 7. Illustrating the Changes in CLs of CVs during Incremental Testing for Identifying Misconfiguration.

6.1.2 Misconfigured Interface for DB Server. In this network configuration scenario, we investigate the presence of a misconfigured interface within the DB server across different network sizes, including various numbers of servers distributed across AS. Here our algorithm identifies the misconfiguration after executing a fixed set of 8 tests in all scenarios of the complex network size with 17 servers distributed across 5 AS. The execution of the algorithm encompasses

Table 9. Elapsed Time (sec) for Misconfigured Interface on DB Server.

Number of tests	Test	Elapsed time (sec)
—	Preparation	0.19
t_1	host $< DNS\ record >$	0.22
—	Preparation	0.24
t_3	nmap $< ip\ addr >$	0.50
—	Preparation	0.50
t_5	ping $-c\ 5 < ip\ addr >$	4.19
—	Preparation	0.07
t_2	ip $-4\ addr\ show\ dev < interface >$	0.15
—	Preparation	0.22
t_4	ifconfig $< interface >$	0.18
—	Preparation	0.13
t_6	traceroute $< ip\ addr >$	0.21
—	Preparation	0.09
t_7	netstat $-I < interface >$	0.13
—	Preparation	0.13
t_8	telnet $< ip\ addr > < port >$	0.12
Total	—	**7.27**

Table 10. DB Server Interface Checks: Assessing CLs of CVs.

CVs	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
IP Address	0.22	0.48	1.00	1.00	1.00	1.00	1.00	1.00
DNS record	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22
Port Num.	0.00	0.26	0.26	0.26	0.26	0.26	0.26	0.41
MTU	0.00	0.00	0.00	-0.06	-0.12	-0.12	-0.12	-0.12
Subnet mask	0.00	0.00	0.00	-0.06	-0.12	-0.12	-0.12	-0.12
Gateway	0.00	0.00	0.00	0.00	-0.06	-0.06	-0.06	-0.06
Interface	0.00	0.00	0.00	-0.06	-0.06	-0.06	-0.42	**-0.42**
Broadcast	0.00	0.00	0.00	-0.06	-0.06	-0.06	-0.06	-0.06

several phases, each contributing to the precise identification of the misconfigured DB server interface. The total elapsed time for the entire procedure is 7.27 seconds, with the crucial aspect being the detection of the misconfigured interface upon completion of the initial 8 tests, as shown in Table 9. While all CVs are above the confidence threshold of -0.6, the algorithm pinpoints the DB server's interface as the most concerning misconfiguration, with a CL of -0.42 (refer to Fig. 7b and Table 10). This highlights the algorithm's ability to identify the most significant misconfiguration, even when individual variables are still above the set threshold.

6.1.3 Misconfigured Subnet Mask for DNS Server. In this scenario, we want to identify a misconfigured subnet mask within the DNS server across different network sizes. Our algorithm detects this in 8 tests in all scenarios. We highlight the scenario in the complex network size. The algorithm's execution is divided into several phases, each contributing to the precise identification of the misconfigured subnet mask within the DNS server. The total elapsed time for the entire process is 7.76 seconds (refer to Table 11), and the critical point is the early detection of the misconfigured subnet mask upon completion of the initial eight tests. Importantly, the CL associated with the subnet mask is -0.12 (refer to Fig. 8 and Table 12).

This scenario sheds light on a crucial aspect of the algorithm's operation: when there are several tests dedicated to checking the misconfiguration (e.g., for the IP address), the algorithm can achieve a high CL for identifying failures. However, when tests involve many CVs or are not sufficiently discriminative, the CL of the true may not have a very low

Table 11. Elapsed Time (sec) for Misconfigured Subnet mask on DNS server

Number of tests	Test	Elapsed time (sec)
—	Preparation	0.18
t_1	host $< DNS\ record >$	0.22
—	Preparation	0.35
t_3	nmap $< ip\ addr >$	0.48
t_5	ping $-c$ 5 $< ip\ addr >$	4.26
—	Preparation	0.24
—	Preparation	0.15
t_2	ip -4 $addr\ show\ dev < interface >$	0.01
—	Preparation	0.33
t_4	ifconfig $< interface >$	0.01
—	Preparation	0.22
t_6	traceroute $< ip\ addr >$	0.19
—	Preparation	0.18
t_7	netstat $-I < interface >$	0.22
—	Preparation	0.23
t_8	telnet $< ip\ addr > < port >$	0.23
Total	—	**7.76**

Table 12. DNS Server Subnet mask Checks: Assessing CLs of CVs.

CVs	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
IP Address	0.22	0.48	1.00	1.00	1.00	1.00	1.00	1.00
DNS record	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22
Port Num.	0.00	0.26	0.26	0.26	0.26	0.26	0.26	0.41
MTU	0.00	0.00	0.00	0.06	0.12	0.12	0.12	0.12
Subnet mask	0.00	0.00	0.00	-0.06	-0.12	-0.12	-0.12	**-0.12**
Gateway	0.00	0.00	0.00	0.00	0.06	0.06	0.06	0.06
Interface	0.00	0.00	0.00	0.06	0.06	0.06	0.42	0.68
Broadcast	0.00	0.00	0.00	0.06	0.06	0.06	0.06	0.06

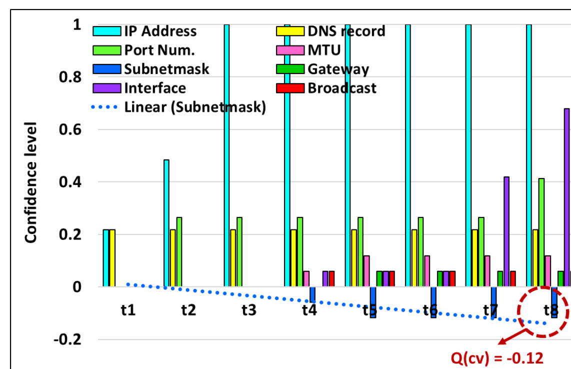
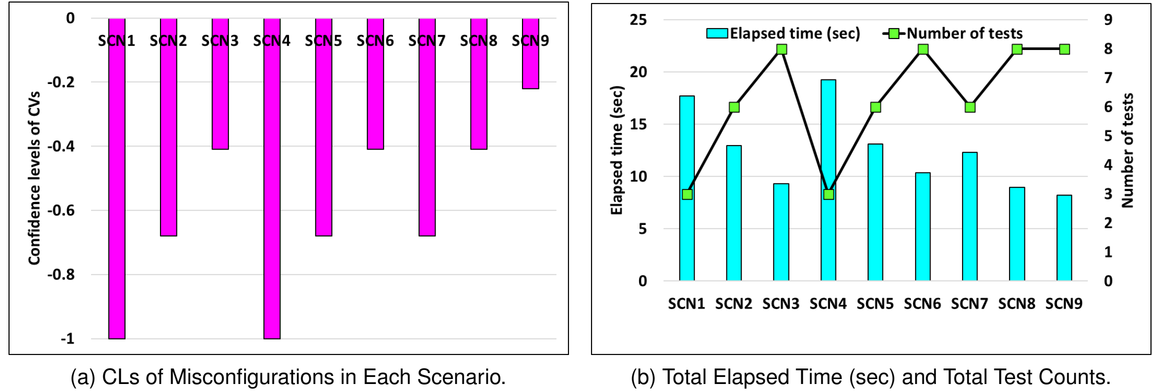


Fig. 8. Changes in CLs of CVs during Incremental Testing for Identifying Incorrect SubnetMask on the DNS Server.

negative value but still has the lowest negative value among other variables. In particular, there are no specific tests for checking the mask; instead, those are typically bundled with checking the IP address. In other words, if we can design a test that focuses on the subnet mask, the algorithm will likely be able to determine its misconfiguration with higher confidence.



(a) CLs of Misconfigurations in Each Scenario.

(b) Total Elapsed Time (sec) and Total Test Counts.

Fig. 9. Overview of scenarios involving misconfigurations in one server.

6.1.4 Misconfigurations in a Single Server. Fig. 9 provides a visual representation of the misconfigurations across various scenarios. The scenarios include misconfigurations in the IP address of the App server (SCN1), port number of the App server (SCN2), the interface of the App server (SCN3), the IP address of the DB server (SCN4), port number of the DB server (SCN5), the interface of the DB server (SCN6), port number of the DNS server (SCN7), the interface of the DNS server (SCN8), and DNS entry of the App server (SCN9). Fig.9a illustrates the CLs associated with each misconfiguration. Notably, scenarios involving IP address misconfigurations in both SCN1 and SCN4 exhibit the most negative CLs, both at -1. Additionally, misconfigurations related to port numbers, whether in SCN2 or SCN5, consistently show CLs of -0.68. Fig.9b represents the elapsed time and the number of tests for each misconfiguration scenario. For instance, scenarios with IP address misconfigurations in both SCN1 and SCN4 show relatively longer elapsed times (17.69s and 19.23s), due to ping unreachability, but require only three tests. Conversely, scenarios like port number misconfigurations in SCN2 and SCN5 strike a balance with moderately low elapsed times (12.98s and 13.09s) and a moderate number of tests (six).

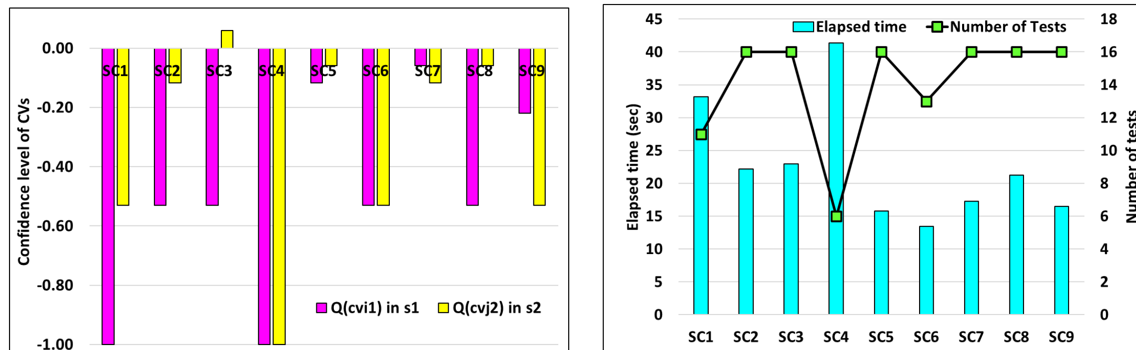
6.1.5 Misconfigurations in Multiple Servers. In our analysis of scenarios, we employ a systematic approach to replicate diverse misconfigurations across network components as shown in Table 13. This table shows 9 scenarios where two services, s_1 and s_2, encounter problems. cv_{i1} and cv_{j2} represent misconfigured CVs in services s_1 and s_2, respectively. In scenario SC1, the App server, named APP_num6, with IP address 10.10.3.30 is supposed to connect to DNS server, named DNS_num50, on port $53/tcp$. However, both the App server and DNS server have different IP addresses and port numbers. This results in a failure between the App server and the DNS server, but the specific misconfigurations causing the fault are unknown. Our approach successfully identifies the problem in both IP address and port number with CLs of $Q(cv_{i1}) = -1$ sand $Q(cv_{j2}) = -0.53$. This information is illustrated in Fig. 10a. Since cv_2 has the most negative CL among the misconfigurations, we prioritize it as the likely cause of the problem.

In scenario SC2, the DNS server, named DNS_num70, and the App server, named APP_num4, are set up with port $53/tcp$ and subnet mask $10.10.3.0/24$, respectively. However, both are misconfigured, causing communication problems between them since they expect different port configurations. Our approach shows CLs ($Q(cvi1) = -0.53$ and $Q(cv_j2) = -0.12$), suggesting a high likelihood of misconfiguration in both the DNS server's port and the App server's subnet mask.

In scenario SC3, the App server APP_num1 is supposed to use port $80/tcp$, and the DB server's gateway (DB_num1) should be set to $10.10.1.2$. However, there is a mistake in their configurations, which can cause a problem in the connection

Table 13. Total Elapsed Time and Total Number of Tests for Diagnosing Two Misconfigurations in Two Different Servers.

Scenarios	s_1	cv_{i1}	s_2	cv_{j2}	$Q(cv_{i1})$	$Q(cv_{j2})$	Num. of Tests	Elapsed time (sec)
SC1	APP_num6	IP Address	DNS_num50	Port Num.	-1	-0.53	11	33.15
SC2	DNS_num70	Port Num.	APP_num4	Subnet mask	-0.53	-0.12	16	22.19
SC3	APP_num1	Port Num.	DB_num1	Gateway	-0.53	0.06	16	22.94
SC4	APP_num6	IP Address	DB_num5	IP Address	-1	-1	6	41.33
SC5	DNS_num50	Subnet mask	DB_num3	Gateway	-0.12	-0.06	18	15.82
SC6	DNS_num60	Port Num.	APP_num8	Port Num.	-0.53	-0.53	13	13.45
SC7	APP_num6	Gateway	DB_num5	Subnet mask	-0.06	-0.12	16	17.23
SC8	APP_num8	Port Num.	DB_num3	Subnet mask	-0.53	-0.06	16	21.23
SC9	DNS_num60	DNS record	APP_num3	Port Num.	-0.22	-0.53	16	16.45



(a) CLs of both Misconfigurations in Each Scenario. (b) Total Elapsed Time (sec) and Total Test Counts.

Fig. 10. Overview of scenarios involving misconfigurations in two services.

between the App server and the DB server. Our approach indicates CLs ($Q(cvi1) = -0.53$ and $Q(cv_{j2}) = -0.06$). These CVs have the most negative CLs compared to other configurations. In SC4, the App server (APP_num6) should use the IP address 10.10.3.30, and the DB server's IP address (DB_num1) is expected to be 10.10.3.60. However, there's an error in their configurations. If the two servers are not intended to communicate directly, this misconfiguration can result in unintended network traffic. Our method reveals high CLs ($Q(cv_{i1}) = -1$ and $Q(cv_{j2}) = -1$) for both misconfigurations.

We also explore scenario SC5 involving conflicting subnet mask and gateway configurations between the DNS server DNS_num50 ($10.1.1.0/24$) and the DB server DB_num3 (10.10.1.2), leading to disrupting network routing between the two servers. Our algorithm assigns the negative CLs ($Q(cv_{i1}) = -0.13$ and $Q(cv_{j2}) = -0.058$) for both the DNS server's subnet mask and the DB server's gateway.

In scenario SC6, the DNS server DNS_num60 is expected to use port number $53/tcp$, and the App server APP_num8 should also be configured with the port number $53/tcp$. LMC shows that there is a mistake in the servers, indicating communication failures, as the two servers are using different ports for communication. The CLs are $Q(cv_{i1}) = -0.53$ for the DNS server's port configuration and $Q(cv_{j2}) = -0.53$ for the App server's port configuration and show misconfigurations correctly.

For scenario SC7, the App server's gateway (APP_num6) is set to 10.10.3.2, and the DB server's subnet mask is configured as $10.10.3.0/24$; however, they are misconfigured. These conflicting configurations can disrupt network routing between the App server and the DB server. Our assessment indicates misconfigurations with CLs ($Q(cv_{i1}) = -0.06$ and $Q(cv_{j2}) = -0.12$) that are the most negative CLs among other CVs.

In SC8, the App server (APP_num8) is expected to use port $80/tcp$, and the DB server (DB_num3) should be configured with a subnet mask $10.10.1.0/24$. However, they are set up wrong, causing problems in how the App server and the DB server connect. our analysis reveals misconfigurations with CLs ($Q(cv_{i1}) = -0.53$ and $Q(cv_{j2}) = -0.06$).

Finally, scenario SC9 involves the DNS server (DNS_num60) with a DNS record ($app3.temple.edu$) and the App server (APP_num3) configured with a port ($80/tcp$). As they are set up wrong, it can cause communication faults as the expected ports do not align. Our approach indicates CLs ($Q(cv_{i1}) = -0.22$ and $Q(cv_{j2}) = -0.53$) for misconfigurations in the DNS server's DNS record and the App server's port number.

Fig. 10b presents the total elapsed time and the number of tests for each scenario. Scenarios like SC1 and SC4, despite longer elapsed times (33.15s and 41.33s), require fewer tests (11 and 6) to achieve high CLs (-1.00 CL), primarily due to ping unreachability. Conversely, scenarios like SC6 show low elapsed time, appropriate CLs, and a low number of tests, indicating rapid identification of misconfigurations.

6.2 LMC Does Not Show Any Failures

In this section, we address a crucial aspect of our network monitoring approach, i.e., where the LMC doesn't flag any faults, yet clients report service problems. We explore three key scenarios: in the following, namely, misconfigured firewall rules, routing failures, and BGP hijacking. In spite of the separate discussion of these in the following, we emphasize that the diagnosis does not assume any knowledge of the type of problem. Rather, the traceroute output itself reveals in determining the problem type as discussed below.

6.2.1 Misconfigured Firewall Rules. To evaluate our diagnosis procedure, we consider three distinct scenarios with different network complexities: small (3 servers in one AS), medium (6 servers in 2 AS), and complex (17 servers in 5 AS). To replicate a firewall-related fault consistently across various scenarios, we implemented specific rules in the router located near the destination in all instances. This ensured uniform conditions across different scenarios and also represents a more difficult case than the one where the problem lies at or close to the source. In the provided example, we utilized 'iptables' rules intentionally designed to induce network problems within the designated path. These (misconfigured) rules are strategically placed in the routers (with integrated firewalls) near the destination. The first rule blocks User Datagram Protocol (UDP) packets from the source IP to the destination IP, while the second rule blocks Internet Control Message Protocol (ICMP) packets of type 'port-unreachable' within the same source-destination pair. By implementing these rules, we simulated a scenario where specific traffic is hindered, allowing us to systematically assess and identify path-related faults in our network monitoring approach.

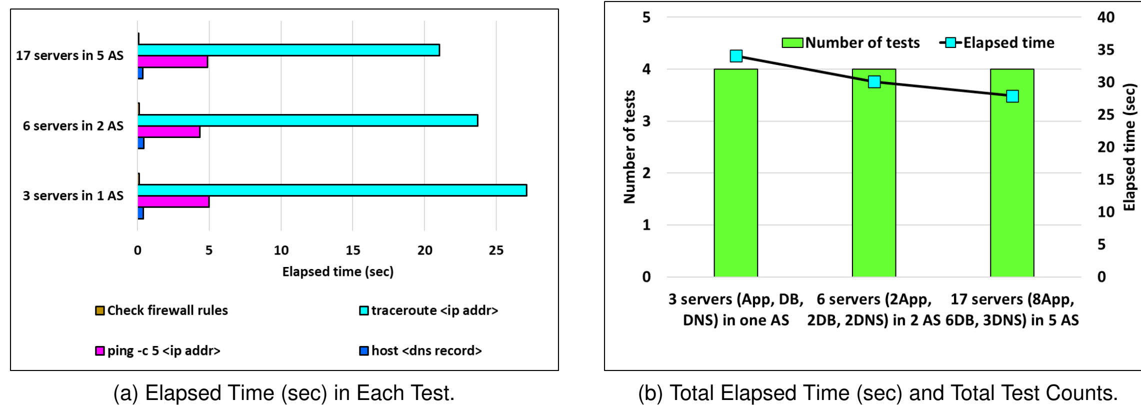
```
iptables -A FORWARD -s 10.1.1.71 -d 10.1.2.20 -p udp -j DROP
iptables -A FORWARD -s 10.1.1.71 -d 10.1.2.20 -p icmp --icmp-type port-unreachable -j DROP
```

The diagnosis uses the following testing hierarchy: We start by checking the name resolution, followed by ping, and then traceroute tests. In the traceroute output (as illustrated in Fig. 11a), we systematically examine the results for our different network sizes. The results, as depicted in Fig. 11b, demonstrate variations in elapsed time across different network sizes. Interestingly, regardless of network size, our approach consistently required only four tests to pinpoint the firewall-related path fault. This illustrates the value of a systematic diagnosis method as opposed to the rather haphazard testing typically done by administrators. It may also be noted that the diagnosis time actually decreases with the size of the network, which appears anomalous. We explain this in the discussion below.

The output snippets for each network size reveal a common pattern:

Small network size: In the small network, the traceroute initially shows successful hops with minimal elapsed time. However, all subsequent hops (beyond hop 1) display '***,' indicating that the traceroute encountered faults reaching the destination after the first hop.

```
1 10.1.3.25 (10.1.3.25)  0.180 ms  0.019 ms  0.014 ms
...
```



(a) Elapsed Time (sec) in Each Test. (b) Total Elapsed Time (sec) and Total Test Counts.

Fig. 11. Misconfigured Firewall rules.

```
30 * * *
```

Medium network size: Similarly, in the medium-sized network, the traceroute begins with successful hops, but '***' appears in subsequent hops, indicating a problem in the path to the destination.

```
1  10.1.4.25 (10.1.4.25)  0.102 ms  0.027 ms  0.016 ms
2  10.50.0.30 (10.50.0.30)  0.045 ms  0.021 ms  0.021 ms
3  10.50.0.25 (10.50.0.25)  0.054 ms  0.036 ms  0.037 ms
...
30 * * *
```

Complex network size: In the complex network, the traceroute exhibits a similar pattern, starting with successful hops but encountering faults in subsequent hops.

```
1  10.1.4.25 (10.1.4.25)  0.090 ms  0.026 ms  0.017 ms
2  10.50.0.30 (10.50.0.30)  0.063 ms  0.030 ms  0.024 ms
3  172.16.100.254 (172.16.100.254)  0.058 ms  0.023 ms  0.023 ms
4  10.50.0.1 (10.50.0.1)  0.053 ms  0.036 ms  0.063 ms
...
30 * * *
```

The abnormal increase in elapsed time and the appearance of '***' in the traceroute output suggest potential disruptions in the network path. Subsequently, we identify the last reachable router (in the complex case, the router at hop 4) and focus on investigating the firewalls within that specific router. This systematic process allows us to pinpoint the location of the firewall-related path fault in the network. The results in Table 14 revealed an intriguing trend: in complex networks, the elapsed time to identify path faults related to firewall rules was comparatively lower. This phenomenon can be attributed to the traceroute output dynamics. *In larger networks, the traceroute command traverses more hops, but the number of hops experiencing timeouts decreases.* Consequently, this impacts the elapsed time for fault identification, making it more efficient in complex network scenarios.

6.2.2 Routing Table Misconfiguration. We investigate routing failures using the traceroute output after conducting host and ping checks. Unlike the firewall-related path faults, where traceroute output displays abnormal elapsed time and '***' in certain hops, routing failures manifest as '!N' or '!H' in the output. The symbol '!N' suggests "network unreachable," and '!H' suggests "host unreachable." For example, let us consider the case of *host*3 with IP address

Table 14. Elapsed Time (sec) for Each Test in Various Scenarios Due to Firewall Rules Problems.

Tests	3 servers in 1 AS	6 servers in 2 AS	17 servers in 5 AS
Preparation	0.65	0.66	0.59
t_1: Host <dns record>	0.421	0.4375	0.38
Preparation	0.29	0.34	0.41
t_2: Ping -c 5 <ip addr>	4.98	4.35	4.89
Preparation	0.35	0.31	0.32
t_3: Traceroute <ip addr>	27.1	23.71	21.03
Jump to the last reachable router	0.10	0.12	0.17
Preparation	0.01	0.01	0.01
t_4: Check firewall rules	0.14	0.13	0.11

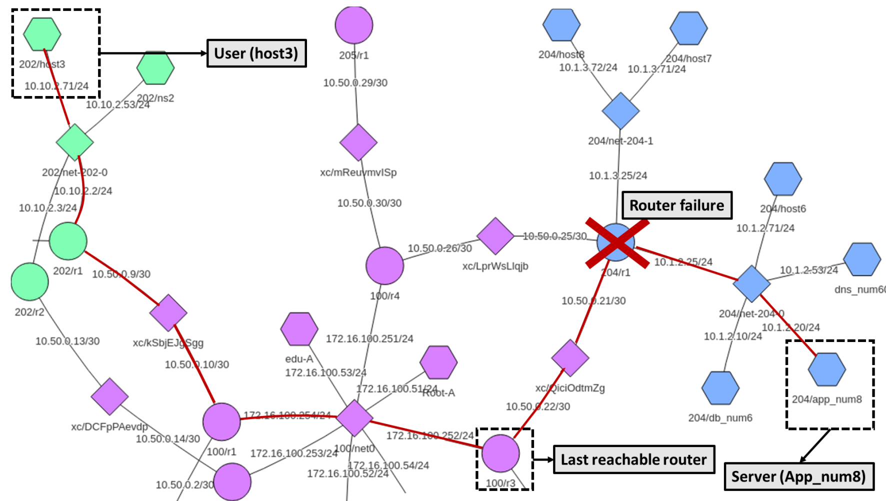


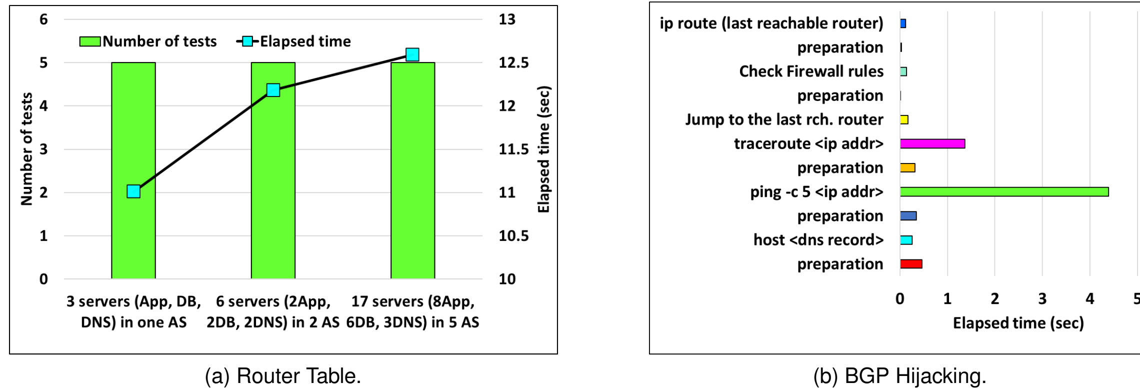
Fig. 12. Snapshot of a Segment in a Complex Network: Router Failure Disrupts the Connection.

10.10.2.71 complaining about a fault connecting to the App server *APP_num8* with IP address 10.1.2.20, as shown in Fig. 12. Despite the complaint, the LMC shows everything is okay. A traceroute to the App server reveals the following output:

```
traceroute to 10.1.2.20 (10.1.2.20), 30 hops max, 60 byte packets
 1  10.10.2.2 (10.10.2.2)  0.150 ms  0.023 ms  0.019 ms
 2  10.50.0.10 (10.50.0.10)  0.185 ms  0.046 ms  0.030 ms
 3  172.16.100.252 (172.16.100.252)  0.278 ms  0.104 ms  0.048 ms
 4  172.16.100.252 (172.16.100.252)  3064.222 ms !H  3064.166 ms !H  3064.129 ms !H
```

In this scenario, the '!H' in the traceroute output indicates a host unreachable condition at hop 4, suggesting a problem with the routing to the App server. Instead of checking firewalls, we jump to the last reachable router identified in the traceroute output, or 172.16.100.252. We then execute the 'ip route' command to inspect the routing table in the last reachable router and identify any changes in its neighbors. We next run the ping command on the neighbor routers since a ping failure would confirm the fault resides with the specific router subjected to the test.

The results in Fig. 13a consistently show that the number of tests required to identify and resolve network faults remains constant at 5 across all scenarios. Additionally, it's noteworthy that the elapsed time exhibits only slight variations,



(a) Router Table. (b) BGP Hijacking.

Fig. 13. a) Router Table Misconfiguration Elapsed Time and Test Count, and b) BGP Hijacking Elapsed Time of Each Test.

Table 15. Elapsed Time (sec) for Each Test in Various Scenarios Due to Router Failure.

Test	3 servers in one AS	6 servers in 2 AS	17 servers in 5 AS
Preparation	0.49	0.56	0.47
t_1: Host <dns record>	0.25	0.19	0.26
Preparation	0.39	0.32	0.35
t_2: Ping -c 5 <ip addr>	4.23	4.53	4.56
Preparation	0.41	0.33	0.32
t_3: Traceroute <ip addr>	0.12	0.92	1.34
Jump to last reachable router	0.21	0.17	0.17
Preparation	0.01	0.01	0.01
t_4: IP route in last reachable router	0.13	0.14	0.12
Preparation	0.23	0.28	0.23
t_5: Ping neighbor router	4.54	4.73	4.76
Total	11.01	12.18	12.59

with the smallest network size taking 11.01 seconds, the medium-sized network taking 12.18 seconds, and the largest, most complex network requiring 12.59 seconds (refer to Table 15).

6.2.3 BGP Hijacking Related Misconfiguration. Consider a prefix hijacking in subnet 10.163.0.0/24 through the exploitation of autonomous system $AS-100$, and suppose that a client complains that the IP address 10.163.0.71 in $AS-163$ is inaccessible from $AS-154$. The traceroute results show abnormal patterns with '***' entries indicating disruptions in the path. The traceroute output looks like this:

```
traceroute to 10.163.0.71 (10.163.0.71), 30 hops max, 60 byte packets
1  as154r-router0-10.154.0.254.output_net_154_net0 (10.154.0.254)  0.169 ms  0.042 ms  0.022 ms
2  10.102.0.2 (10.102.0.2)  0.067 ms  0.070 ms  0.052 ms
3  10.2.1.254 (10.2.1.254)  0.091 ms  0.062 ms  0.067 ms
4  * * *
...
30 * * *
```

In this scenario, the last reachable router is identified as 10.2.1.254, which is the neighbor of the router used for network hijacking on 10.163.0.0/24, as shown in Fig. 14. In contrast to the situations involving routing failures, where firewall

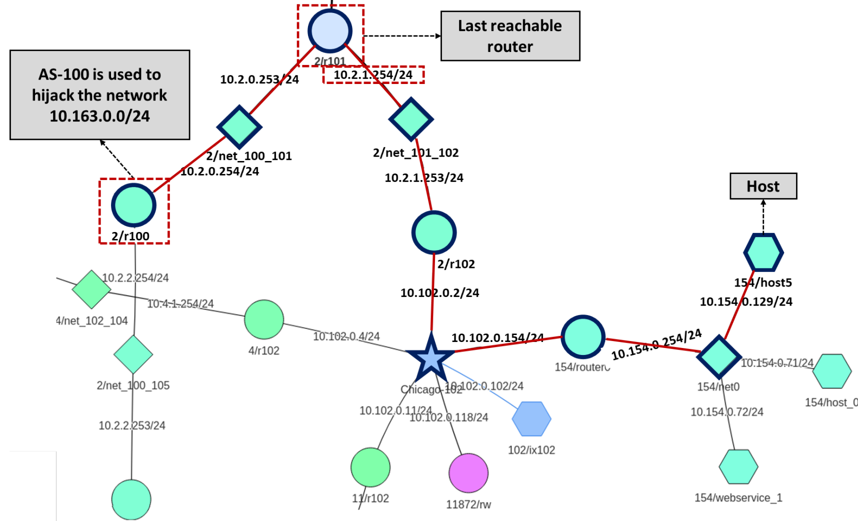


Fig. 14. Snapshot of a Segment hijacked by $AS - 100$.

checks may be unnecessary, BGP hijacking incidents necessitate an examination of firewall rules due to abnormalities marked by '***' in the traceroute output.

To investigate further, we examine the neighboring routers of 10.2.1.254 using the 'iproute' command. By closely examining the routing table of 10.2.1.254, potential black hole entries or irregularities can be identified. In this case, specific attention is given to the entry related to 10.163.0.0/24. The investigation then extends to the router that advertised this route to 10.2.1.254. By confirming the route propagation and checking for any signs of a black hole, the analysis aims to pinpoint the source of the BGP Hijacking. The total elapsed time for these BGP hijacking tests, conducted in a complex network with 5 AS, is 7.62 seconds, as shown in Fig. 13b.

6.3 Tuning Thresholds

Our diagnosis procedure involves some thresholds that provide a trade-off between the number of tests needed to diagnose the problem and the chances of misdiagnosis. The three most relevant parameters are as follows:

Misconfiguration Threshold ($MT \in [-1..+1]$), which is the threshold value for the CL of the CVs. In general, there are two of these: (a) upper threshold MT_h, such that if the CL exceeds MT_h, we declare the CV as set properly, and lower threshold MT_l, which is a negative number, so that if the CL of a CV drops below MT_l, we declare this CV to be contributing to the observed problem. In practice, we find that $MT_h = 0.5$ is adequate in all cases and does not need to be tuned. Thus, MT really refers to MT_l. Our experiments show that $MT_l = -0.6$ provides a good compromise between test expense and false positives.

Number of CVs involved in the test whose CL can remain zero (ZC). Recall that the CL of all CVs relevant to a service is initially set to zero. Some subset of these CVs will be involved in each test, and thus their CL will be zero initially. As the testing proceeds, some of the CL of some of the service CVs may be set to positive or negative values while others remain zero. The threshold ZC essentially controls whether we go for exploring the CVs about which do not know anything (i.e., those with a CL of zero) or try to further explore those that already have a significant positive or negative value, in the hope of driving them beyond the MT threshold and thus disposing them off. Note that most tests will involve only a few CVs, therefore specifying ZC as an absolute number (as opposed to a fraction of the total number of CVs involved) is adequate and preferred.

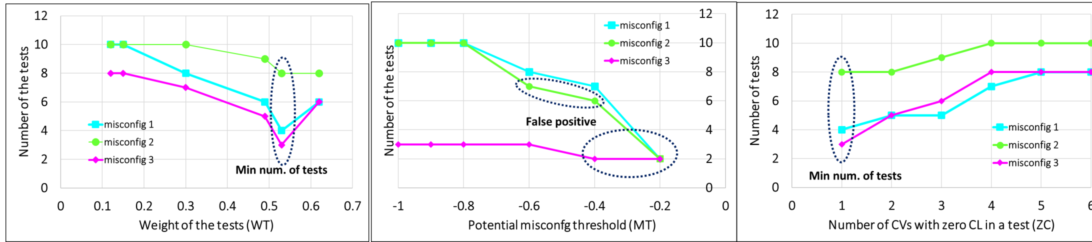


Fig. 15. Balancing Act-Optimal Thresholds for Efficient Misconfiguration Detection. a) Fixed MT = -0.6 and ZC = 1, variable WT; b) Fixed ZC = 1 and WT = 0.5, variable MT; and c) Fixed MT = -0.6 and WT = 0.5, variable ZC.

Weight of the Test ($WT \in 0..1$), which indicates how revealing the test is on the average. The true WT is obviously a function of the specific problem, which is what we are trying to diagnose. We resolve this dilemma by enumerating the major problem types and assigning a WT value based on domain knowledge. Such assignment must be easy for administrators, and thus, we assign only 3 possible values to a test relative to a specific problem: 0 (the test is irrelevant to the problem), 0.5 (the test is somewhat relevant), and 1.0 (the test is highly relevant). The overall WT could then be obtained as average over all the considered problems and thus merely represents an overall quality for the test without being tied to any specific fault.

To optimize these thresholds, we conducted a systematic grid search aimed at minimizing the number of tests while ensuring precise misconfiguration detection (as illustrated in Figure 15). While we conducted the search for various scenarios, we highlighted the outcomes for three specific cases. The results indicate that for WT, if set too low (e.g., 0.12), the algorithm may excessively focus on the count of CVs with zero CLs, potentially neglecting tests with higher weights that are still significant. Conversely, if set too high (e.g., >= 0.6), the algorithm may overly prioritize test weights, neglecting tests with a good balance between the weight and the count of CVs with zero CLs. Such tests will fail to reduce the number of CVs with zero CL and thus lead to an increase in the overall number of conducted tests. Regarding MT, if set too high (say, -0.1), the algorithm may stop prematurely, resulting in false positives in most cases. On the other hand, if set too low (say, -0.9), the algorithm may take longer to converge, conducting unnecessary tests for identifying misconfigurations. To address this, we compute the false positive ratio at different MT values for each CV. The False Positive (FP) and precision formulas are as follows:

$$\text{False Positive} = \frac{\text{Number of False Positives}}{\text{Total Predicted Positives}}, \qquad \text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

The precision for $MT \leq -0.6$ is 0.88, and we designate this as our threshold. For ZC, a low setting (e.g., 1) can result in the algorithm choosing tests with at least one zero CL but high weight, such as ping and nmap. This ensures that these tests are conducted earlier, as indicated in line 3 of Algorithm 2. Conversely, when we raise the ZC threshold (e.g., 5), tests with high weight and few zero counts may not satisfy the condition, leading to a delay in conducting these crucial tests. Our study indicates that the best values for thresholds are MT at -0.6, ZC at 1, and WT at 0.5.

7 CONCLUSION AND FUTURE WORK

In this paper, we have addressed the problem of diagnosing misconfigurations in enterprise IT systems. The proposed root-cause analysis framework focuses on reactive diagnosis, sequentially selecting diagnostic tests to minimize testing effort. We show that many of the commonly occurring misconfiguration problems can be tackled automatically without any manual intervention. We saw that the root-cause of the problem takes less than 35 seconds in all cases. This is to be compared against the typical manual process where the administrator might try different tests with diagnosis times

ranging from minutes to hours. Furthermore, since we only use the universally available test commands, the mechanism can be easily implemented in any enterprise network. We plan to open-source our solution so that it can be used and further enhanced by the community.

Our effort so far made several simplifying assumptions that we plan to address in the future. The most prominent of these is the assumption of a global console which we discuss below. Other aspects relate to the limitations of the SEED emulator in its current state, but given the flexible architecture, it is possible to extend SEED for many additional capabilities, as we demonstrated with VRRP. In particular, SEED currently does not support layer2 networks but can be implemented by using open-source switching software like OPX. Another issue concerns our assumption of visibility being limited to a subnet that has its own test agent. However, defining visibility boundaries and, hence, the scope of LMC across multiple subnets is straightforward.

7.1 Limitations of Diagnosis Methodology

It is important to note that our diagnosis methodology is not intended to encompass the deep semantics of complex enterprise applications; instead, it is largely intended for interactions between applications through the network and their basic operability status. Thus the only configuration variables of interest to our algorithm are those relating to the reachability of a service and whether the basic parameters of the service (e.g., DNS mapping, firewall settings, routing, etc.) are set up properly or not. In case the service is found to be nonresponsive or ill configured by these very basic criteria, the responsibility of detailed debugging shifts to other, focused tools that can deal with the complexities of the application. From this perspective, *the foundational principles of our model are adaptable and can be extended to other applications and for somewhat deeper examination of the applications that we have already considered.* The methodology would thus lead to automated root-causing up to the level of a small set of individual services, which could then be examined further.

As an example, DNS itself has many aspects to its configuration that we have not touched upon. In particular, DNS allows records for specific services (e.g., email), and such records have a priority as a configuration parameter. Now if the Mail eXchange (MX) record with the smallest priority value (which is actually the highest priority) points to a mail server that does not accept Simple Mail Transfer Protocol (SMTP) connections, the mail send will fail over to the other exchange servers. Diagnosing such a problem would require tests to check the mail server configuration, too. Similarly, a missing pointer (PTR) record (reverse translation) may flag the incoming mail as spam. Suitable tests, along with their weight, need to be added to handle such features.

7.2 Dependency Characterization

In this work, we have assumed that the dependencies between services and their CVs are known explicitly from the description of the services and the domain knowledge. However, there are invariably some dependencies that are not known due to the lack of good documentation. Or, they may not be obvious if they arise only in special situations such as a failure or even as a result of misconfiguration [35]. The discovery of hidden dependencies is a complex topic that we have not pursued in this paper. Several methods have been proposed in the literature to explore dependencies [36–38]. However, automated methods may not expose all dependencies or identify some false dependencies. Maintaining a database of discovered or suspected dependencies can be helpful in reducing such situations but cannot eliminate them. The impact of unknown dependencies is that a faulty service that does not show any abnormal behavior but affects others may not be selected for testing. This would likely extend the testing campaign until many others have been tested unnecessarily and the attention turns to the untested ones.

7.3 Advancing Testing Methodologies

One offshoot of the proposed methodology is the insights that it provides into the tests themselves. As noted in section 6.1, with the existing tests, it is difficult to determine if the port or the gateway is misconfigured; thus, an additional test that can distinguish between those two misconfigurations will be highly valuable. This applies in general – the ability to root-cause a problem very much depends on the discriminative power of the tests and has implications for the test design. Formally speaking, consider the test set $T_R(s_i)$ that is relevant for service s_i with $CV_i = \{cv_{i1}, cv_{i2}, \ldots, cv_{ik_i}\}$ as its k_i CVs. In general, any of these CVs (or their combination) could be misconfigured. Thus to enable identification of any misconfigured CV, say cv_{ij}, $T_R(s_i)$ must be such that after running all of the tests, we have: $CL(cv_{ij}) < CL(cv_{ik})$ for all k. Since we already have a base of existing tests, additional tests should be designed to provide the maximal disambiguation subject to feasibility and other constraints on the tests.

7.4 Global Console Emulation

As mentioned before, the abstraction of a global console must deal with the unavoidable distributed systems issues. In particular, consolidating status from multiple LMCs will involve varying delays on top of the varying delays in reflecting status changes in the LMC (usually very small) and delays in remote access to this information. As is well recognized, it is impossible to learn the true status of remote entities in a distributed system. Furthermore, very frequent or entirely event-driven status transmission from all LMCs to a centralized GMC site is unscalable. With permanent and relatively infrequent faults, a periodic fetching of LMC status along with retry in case of incorrect diagnosis should be adequate; however, other methods, such as hierarchical diagnosis, may be needed in other cases and will be explored in the future.

Another major issue in a large distributed system is that of complex accessibility and visibility issues, even for testing purposes. In particular, even the Points of Presence (PoPs) of a single geographically distributed organization may be unwilling to allow remote testing or restrict what types of tests are allowed. This substantially complicates testing and may require testing from specific testing agents. We have examined the issue of accessibility in the past [27, 39, 40], but its integration into a comprehensive testing strategy remains.

REFERENCES

[1] S. Newman, *Building microservices*. " O'Reilly Media, Inc.", 2021.

[2] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[3] M. Williams and A. Vance, "Microsoft takes blame for web site access failures," URL http://www.computerworld.com/article/2590639/networking/microsoft-takes-blame-for-web-site-access-failures.html, 2001, [Online; accessed 3-July-2023].

[4] "Misconfiguration brings down entire .se domain in sweden," URL http://www.circleid.com/posts/misconfiguration_brings_down_entire_se_domain_in_sweden, 2009, [Online; accessed 3-July-2023].

[5] "Apple blames itunes outage on dns error. what does that mean?" URL https://www.csmonitor.com/Technology/2015/0311/Apple-blames-iTunes-outage-on-DNS-error.-What-does-that-mean, 2015, [Online; accessed 3-July-2023].

[6] A. Kurtz, "Delta malfunction on land keeps a fleet of planes from the sky," URL https://www.nytimes.com/2016/08/09/business/delta-air-lines-delays-computer-failure.html, 2016, [Online; accessed 3-July-2023].

[7] "Southwest airlines' router grounds 2,300 flights," URL https://availabilitydigest.com/public_articles/1108/southwest_airlines.pdf, 2016, [Online; accessed 3-July-2023].

[8] W. Du, H. Zeng, and K. Won, "Seed emulator: an internet emulator for research and education," in *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, 2022, pp. 101–107.

[9] M. Brodie, I. Rish, and S. Ma, "Optimizing probe selectionfor fault localization," in *Proceedings of the 12th IFIP/IEEEInternational Workshop on Distributed Systems: Operationsand Management (DSOM01)*, 2001.

[10] I. Rish, M. Brodie, N. Odintsova, S. Ma, and G. Grabarnik, "Real-time problem determination in distributed systems using active probing," in *2004 IEEE/IFIP Network Operations and Management Symposium (IEEE Cat. No. 04CH37507)*, vol. 1. IEEE, 2004, pp. 133–146.

[11] I. Rish, M. Brodie, S. Ma, N. Odintsova, A. Beygelzimer, G. Grabarnik, and K. Hernandez, "Adaptive diagnosis in distributed systems," *IEEE Transactions on neural networks*, vol. 16, no. 5, pp. 1088–1109, 2005.

[12] A. X. Zheng and I. Rish, “Efficient test selection in active diagnosis via entropy approximation,” *arXiv preprint arXiv:1207.1418*, 2012.

[13] M. Natu and A. S. Sethi, “Active probing approach for fault localization in computer networks,” in *2006 4th IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services*. IEEE, 2006, pp. 25–33.

[14] ——, “Probabilistic fault diagnosis using adaptive probing,” in *International Workshop on Distributed Systems: Operations and Management*. Springer, 2007, pp. 38–49.

[15] ——, “Efficient probing techniques for fault diagnosis,” in *Second International Conference on Internet Monitoring and Protection (ICIMP 2007)*. IEEE, 2007, pp. 20–20.

[16] ——, “Efficient probe selection algorithms for fault diagnosis,” *Telecommunication systems*, vol. 37, pp. 109–125, 2008.

[17] ——, “Probe station placement for fault diagnosis,” in *IEEE GLOBECOM 2007-IEEE Global Telecommunications Conference*. IEEE, 2007, pp. 113–117.

[18] D. Jeswani, M. Natu, and R. K. Ghosh, “Adaptive monitoring: application of probing to adapt passive monitoring,” *Journal of Network and Systems Management*, vol. 23, pp. 950–977, 2015.

[19] L. Lu, Z. Xu, W. Wang, and Y. Sun, “A new fault detection method for computer networks,” *Reliability Engineering & System Safety*, vol. 114, pp. 45–51, 2013.

[20] Y. Tang, E. S. Al-Shaer, and R. Boutaba, “Active integrated fault localization in communication networks,” in *2005 9th IFIP/IEEE International Symposium on Integrated Network Management, 2005. IM 2005.* IEEE, 2005, pp. 543–556.

[21] R. do Carmo, J. Hoffmann, V. Willert, and M. Hollick, “Making active-probing-based network intrusion detection in wireless multihop networks practical: A bayesian inference approach to probe selection,” in *39th Annual IEEE Conference on LCN*. IEEE, 2014, pp. 345–353.

[22] M. S. Garshasbi, “Fault localization based on combines active and passive measurements in computer networks by ant colony optimization,” *Reliability Engineering & System Safety*, vol. 152, pp. 205–212, 2016.

[23] B. Patil, S. Kinger, and V. K. Pathak, “Probe station placement algorithm for probe set reduction in network fault localization,” in *2013 International Conference on Information Systems and Computer Networks*. IEEE, 2013, pp. 164–169.

[24] E. Salhi, S. Lahoud, and B. Cousin, “Localization of single link-level network anomalies,” in *2012 21st International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2012, pp. 1–9.

[25] S. Traverso, E. Tego, E. Kowallik, S. Raffaglio, A. Fregosi, M. Mellia, and F. Matera, “Exploiting hybrid measurements for network troubleshooting,” in *2014 16th International Telecommunications Network Strategy and Planning Symposium (Networks)*. IEEE, 2014, pp. 1–6.

[26] A. Dusia and A. S. Sethi, “Recent advances in fault localization in computer networks,” *IEEE Communications Surveys & Tutorials*, vol. 18, no. 4, pp. 3030–3051, 2016.

[27] I. El-Shekeil, A. Pal, and K. Kant, “CloudMiner: A systematic failure diagnosis framework in enterprise cloud environments,” *Proc. of CLOUDCOM, Nicosia, Greece*, Dec 2018.

[28] S.-M. Lamraoui and S. Nakajima, “A formula-based approach for automatic fault localization of multi-fault programs,” *Journal of Information Processing*, vol. 24, no. 1, pp. 88–98, 2016.

[29] C. Chen, H. Yu, Z. Lei, J. Li, S. Ren, T. Zhang, S. Hu, J. Wang, and W. Shi, “Balance: Bayesian linear attribution for root cause localization,” *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–26, 2023.

[30] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[31] M. Zhang, Z. Li, B. Dahhou, M. Cabassud, and C. Volosencu, “Root cause analysis of actuator fault,” in *Actuators*. IntechOpen, 2018, p. 131.

[32] P.-W. Tsai, F. Piccialli, C.-W. Tsai, M.-Y. Luo, and C.-S. Yang, “Control frameworks in network emulation testbeds: A survey,” *Journal of computational science*, vol. 22, pp. 148–161, 2017.

[33] J. Ahrenholz, C. Danilov, T. R. Henderson, and J. H. Kim, “CORE: A real-time network emulator,” in *MILCOM 2008-2008 IEEE Military Communications Conference*. IEEE, 2008, pp. 1–7.

[34] W. Du and H. Zeng, “The seed internet emulator and its applications in cybersecurity education,” 2022.

[35] R. Vanciu and V. Rajlich, “Hidden dependencies in software systems,” in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[36] D. Strasunskas and S. E. Hakkarainen, “Domain model-driven software engineering: A method for discovery of dependency links,” *Information and Software Technology*, vol. 54, no. 11, pp. 1239–1249, 2012.

[37] H. Zhang, J. Li, L. Zhu, R. Jeffery, Y. Liu, Q. Wang, and M. Li, “Investigating dependencies in software requirements for change propagation analysis,” *Information and Software Technology*, vol. 56, no. 1, pp. 40–53, 2014.

[38] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, “Understanding and discovering software configuration dependencies in cloud and datacenter systems,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 362–374.

[39] M. . Athamnah, A. Pal, and K. Kant, “A framework for misconfiguration diagnosis in interconnected multi-party systems,” *Proc. of ICCCN 2018*, 2018.

[40] M. Athamnah and K. Kant, “Multiparty database sharing with generalized access rules,” in *Proc. of CloudCom, Luxemburg*, Dec 2016, pp. 198–205.