CloudMiner: A Systematic Failure Diagnosis Framework in Enterprise Cloud Environments

Ibrahim El-Shekeil, Amitangshu Pal, Krishna Kant
Computer and Information Sciences, Temple University, Philadelphia, PA 19122
Email: {ielshekeil,amitangshu.pal,kkant}@temple.edu

***Abstract*—Applications and network services in enterprise cloud environments have direct and indirect dependencies. The configuration of these services varies based on business needs. However, accurate and complete documentation of the configuration may not exist at all times. Thus, failure diagnosis becomes further complex with such unknown/uncertain dependencies. To cope with this, some probing stations need to be installed in suitable locations in the network to provide full monitoring and diagnosing capability. In this paper we develop a novel `CloudMiner` architecture for failure diagnosis in enterprise clouds that consist of developing intelligent probing station selection, failure detection and diagnosis across the network components using the minimum set of network probes, considering the inter-dependencies across the network services/components. Extensive simulation results show that `CloudMiner` can always identify the faulty components among the list of a small set of suspected components, the size of which is as low as $\sim$3 for a network with 460 components.**

I. Introduction

A. Overview

Recently more and more enterprises are moving their applications to cloud for flexibility, scalability and lower cost [1]. The increasing dependence on computerized cloud services requires the operators to keep the resources operational and responsive at all times. Guaranteeing availability and services is thus a big challenge for cloud providers. Basic network services are often specified a lofty availability target of five 9s, i.e., 99.999 percent or more availability; which means a downtime of less than 5.25 minutes per year. Unfortunately, downtime incidents commonly violate such goals.

The key difficulty in achieving high availability is an extensive set of dependencies for any given service, and continued increase in these dependencies as services become more sophisticated. Large cloud providers have hundreds and thousands of applications and services. The operations of an application may depend on multiple network services and components spanning multiple virtual and physical hosts and complex dependencies among these services. The identification of these dependencies is non-trivial, error prone and may take long time [2], [3]. Therefore, accurate and complete documentation of the configuration may not exist at all times. Failure detection and localization become further complex with the increase of the complexity of the configuration and with the lack of proper documentation. As a result, the dependencies are often not known or fully understood and may come into play under certain circumstances.

For a service to work properly, all the software and hardware components that it depends on must also work properly. Thus the increasing dependencies not only increase the chances of malfunction but also expand the associated downtime because of the difficulty in identifying which component is faulty and in what way. This makes the task of fault diagnosis very difficult and time consuming, and hence hinders in achieving the high availability requirements.

This research was supported by the NSF grant CCF-1407876.

Towards this end, we propose a systematic failure diagnosis framework in an enterprise cloud environment, named `CloudMiner` that can proactively discover the likelihood that a network component is faulty or misconfigured. However, since we do not know a priori where the problem is, an intelligent selection of probes is crucial subject to suitable access constraints. These represent the key research aspects for making our `CloudMiner` work effectively. Fault diagnosis comprises of two phases, monitoring to detect the misbehavior and troubleshooting to diagnose and localize the fault. We define a *probe* as a testing transaction that interacts with multiple components and has a result of pass or fail. A *diagnosis* is the analysis of the results of a collection of probes. Some simple examples of probes are the ICMP "ping" message and the trace-route to test the connectivity and reachability of nodes in the network or a HTTP/SOAP request to test the service functionality. The probes are sent from probing stations in various locations of the network. One drawback of probing is the additional generated traffic in the network in which has to be minimized. In general, the testing may require multiple probes launched from multiple "probing stations". The proper location of probing stations is also crucial since the location determines what kinds of probes can be launched and what they can access.

To this end the key contributions of `CloudMiner` are as follows. First, we devise an efficient algorithm to select minimum number of probing stations that can monitor all components in the network. Second, we generate a minimum set of probes that the `CloudMiner` needs to run occasionally to perform detection and diagnosis of the faults in an enterprise cloud environment. `CloudMiner` does this by modeling the unknown dependencies across the network components using *uncertainty* theory [4], which is a novel alternative theory of measuring indeterminacy. Third, we perform extensive simulations that show that `CloudMiner` provides a set of suspected components which consists of the actual faulty component. The size of this suspected set is as small as $\sim$3 for a network size of 460 components.

B. Related Work

Various approaches have been proposed in previous works on IP network failure detection and localization. Ozmutlu et. al. [5] studied the probe selection problem for network monitoring to predict the delay between a source and a destination as well as to identify anomalies in a network. Zheng et. al. [6]

studied the probe selection problem for detecting large-scale router failures and localizing the failed routers. Rish et al. [7], [8] proposed some heuristic based approaches for probes set selection for detection and localization of failure in distributed systems and adaptive probe selection approach to select new probes based on the results of previous ran probes.

Various works have been proposed to solve the testing station selection problem. References [9] and [10] use explicitly routed probes to reduce the number of required testing stations. However, to perform this, a source routing must be enabled in the network which is usually disabled to prevent IP source route attacks. Jeswani, Deepak, et al. [11] solved the minimum testing station selection problem by a systematic reduction to the Minimum Hitting Set problem but assumed static single-path routing which is not practical in large networks that mostly use dynamic routing and multiple paths for resiliency.

Different problems have been investigated using the uncertainty theory which was founded by Liu [4] in 2007 and subsequently studied by many researchers. Han et al. [12] investigated maximum flow problem. Zhang and Peng [13] discuss uncertain optimal assignment problem in which the profit is uncertain. Chen et al. [14] present an uncertain programming model for minimum weight vertex covering problem.

In this paper, we assume that the network topology is known and uses dynamic routing that selects shortest paths, and we also assume that the source routing is disabled. We rank potential fault locations similar to the approach that is used in software testing by Jones and Harrold [15]. Our approach addressed the diagnosis of network nodes, links and services. All of the previous works assume that the services dependencies are known which is not realistic in practice, since the configurations are usually unknown, largely due to slow drift of configurations from initial known configurations [16].

C. Paper Organization

The remainder of the paper is organized as follows: Section II describes different types of network service dependencies and presents concepts about uncertainty theory to model these unknown dependencies. Section III provides the detail overview on our `CloudMiner` framework. Section IV presents our detailed experimental evaluations. We conclude the paper in Section V.

II. Preliminaries

In this section, we explain the service dependencies and provide examples. Then, we briefly state some fundamental concepts about uncertainty theory, which are excerpted from Liu [4] and about submodular functions which will be used throughout the paper.

A. Dependency Characterization

Most systems have a hierarchy of services, with higher level services built on top of, and therefore, dependent upon, lower level services. For example, a web application may rely on many supporting services, such as Domain Name System (DNS), Active Directory (AD) and Kerberos, and database (DB). A failure of any supporting service would cause requests to the web application to fail.

A higher level service such as a payroll service would comprise of a web server, application, and a database. All of these components must be installed and configured correctly for the service to run correctly. Moreover, other system and network settings should be configured correctly as well, such as a server(s) operating systems, DNS records, firewall and NAT (network address translation) rules. Another example is MS SharePoint. Fig. 1 shows a possible configuration of MS SharePoint [1]. It comprises of web servers, application servers, and database servers. Based on the size of the installation, it may also require dedicated servers for search components. Additionally, the SharePoint and many other services in a Microsoft environment heavily depend on Microsoft AD (active directory) for user authentication and authorization. Furthermore, each service requires computing, storage, and network resources. Some of these components are direct dependencies while others are indirect dependencies. All components of a service, and its hosting server(s) and the network services must be running and configured correctly; otherwise, the service will not satisfy the user needs.

Following [3], we represent a dependency as $A \rightarrow B$, where A is a depending service and B is a depended service. A service (or an application) may have direct and indirect dependencies. A *direct dependency* is required by a service/application to perform its functionality such as a web application For example a service A retrieves data from the database B. If the database stops working, the application will fail to return the required information to the client. An *indirect dependency* is required by the client accessing the service/application such as resolving the URL name of a web application (service) at the DNS or passing through a firewall or a proxy server to reach the service. The client can access A after it has successfully accessed B. In [17], the authors classified the *direct dependencies* as local-remote and *indirect dependencies* as remote-remote.

As an example, Fig. 2 shows an example of service dependencies. Let's assume that the node B is an AD domain controller server which provides network services such as DNS and authentication. When a client A accesses an application C, it will resolve C's name and authenticate at B after that it can access C, and then C would resolve D's name at B and then access and retrieve or update data in the DB at D. In this case, there are indirect dependencies $C \rightarrow B$ and $D \rightarrow B$ and a direct dependency is $C \rightarrow D$. Notice here that B is a depended service for both C and D, hence it has specific settings for each one of them. In other words, malfunctioning of B could affect both C and D or only one of them; C or D. In the rest of the paper we use service and application interchangeably.

The dependencies are not always as simple as shown in Fig. 2. For instance, in a Microsoft network environment, the AD has logical and physical structures. The logical structure includes forest, domains, child-domains, organizational units and global catalogs. The physical part includes domain controllers and sites (locations in which the domain controllers run). There is no correlation between these two structures. The AD Domain Services require a DNS service, and it will automatically create a DNS delegation and configuration when creating a

[1]Technical diagrams for SharePoint 2013, https://technet.microsoft.com/en-us/library/cc263199.aspx, accessed on May 31st, 2017

TABLE I. AD SERVICES WHICH ARE RUNNING ON THE HOST PHILADELPHIA.EXAMPLE.COM

_Service._Protocol.DnsDomainName	Priority	Weight	Port Num	Target host
_ldap._tcp.example.com	0	0	389	philadelphia.example.com
_kerberos._tcp.example.com	0	0	88	philadelphia.example.com
_ldap._tcp.dc._msdcs.example.com	0	0	389	philadelphia.example.com
_kerberos._tcp.dc._msdcs.example.com	0	0	88	philadelphia.example.com

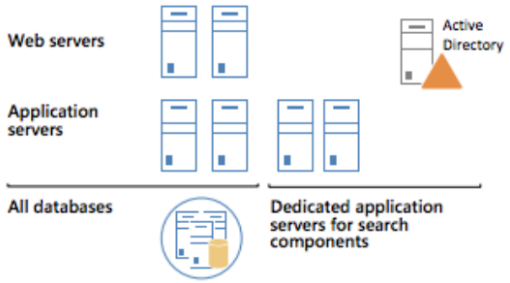


Fig. 1. MS SharePoint Components

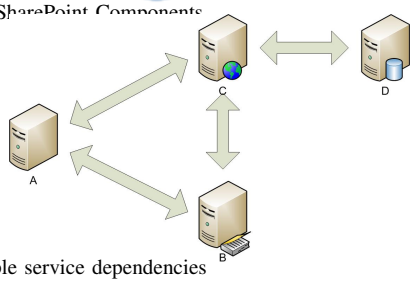


Fig. 2. Simple service dependencies

new domain. Based on the AD size and number of users and services in the environment, multiple domain controllers may be required in each domain, and multiple domains and child-domains may exist. The DNS and the various AD services may or may not run on the same hosts. In large enterprise networks and data centers, identifying the services and its dependencies is non-trivial.

The following example illustrates the combined information that is contained in DNS records of a domain controller named Philadelphia in the domain example.com and has an IP address of 10.0.0.1. It registers in the DNS the following A record `philadelphia.example.com A 10.0.0.1` and SRV records in Table I. Any misconfiguration to any of these records would have an impact on the associated services reachability and to any other service that depends on it. Other complex dependency scenarios may exist with services in load balancing and load sharing configurations and in security configurations.

B. Modeling Indeterminacy in Dependencies

Definition 1 (Indeterminacy): Indeterminacy is the phenomena whose outcomes cannot be predicted in advance. Tossing a coin, rolling a dice or stock prices are real examples of indeterminacy. In order to deal with an indeterminate quantity, there exists two mathematical systems, namely *probability theory* and *uncertainty theory*.

Definition 2 (Probability Theory): A probability of an event with finitely many outcomes is the number of outcomes favorable to that event, divided by the total number of outcomes. Thus the probability theory deals with the *frequency* of outcomes of the event.

Definition 3 (Uncertainty Theory): A fundamental premise of applying probability theory is the fact that the estimated probability is close to the long-running frequency. Probability theory is no longer applicable if the law of large number is not valid. In reality, we are very often lack of observed data in regards to a real event. That is where the uncertainty theory comes into the picture. In such cases we need to take some expert's view on the outcome of an event, which we define as *belief*. A belief degree is defined by the strength of someone's belief that the event will happen. A belief degree is assigned a number between 0 and 1: higher the belief degree is, more strongly someone believes that the event will happen. Thus the belief degree is the empirical basis of uncertainty theory, as opposed to frequency of outcomes in case of probability theory.

Definition 4 (σ-algebra [4]): Let Γ be a non-empty set. A collection $\mathcal{L}$ of subsets of Γ is called a σ-algebra if (a) $\Gamma \in \Lambda$, (b) if $\Lambda \in \mathcal{L}$ then $\Lambda^c \in \mathcal{L}$, and (c) if $\Lambda_1, \Lambda_2, \ldots, \Lambda_n \in \mathcal{L}$, then $\bigcup_{i=1}^{n} \Lambda_i \in \mathcal{L}$.

Definition 5 (Uncertainty space): Let Γ be a nonempty set, and $\mathcal{L}$ be a σ-algebra over Γ, and $\mathcal{M}$ be an uncertain measure. Then the triplet $(\Gamma, \mathcal{L}, \mathcal{M})$ is called an uncertainty space.

Definition 6 (Uncertain measure): Let Γ be a nonempty set, and let $\mathcal{L}$ be a σ-algebra over Γ. Each member $\Lambda \in \mathcal{L}$ is called an event. A set function $\mathcal{M} : \mathcal{L} \to [0, 1]$ is said to be uncertain measure if it satisfies the following four axioms:

Axiom 1: (Normality Axiom) $\mathcal{M}\{\phi\} = 0$ (for null set ϕ) and $\mathcal{M}\{\Gamma\} = 1$ (for the universal set Γ).

Axiom 2: (Duality Axiom) $\mathcal{M}\{\Lambda\} + \mathcal{M}\{\Lambda^c\} = 1$ for any event Λ.

Axiom 3: (Subadditivity Axiom) For every countable sequence of events $\Lambda_1, \Lambda_2, \ldots$, we have

$$\mathcal{M}\left\{\bigcup_{k=1}^{\infty} \Lambda_k\right\} \leq \sum_{k=1}^{\infty} \mathcal{M}_k\{\Lambda_k\}$$

These 3 axioms are intuitive and are satisfied by the probability theory as well. The next axiom, which relates to the product space of multiple uncertainty spaces. That is, if Γ_i is the i-th uncertainty space ($i = 1 \ldots n$), then the product $\Gamma = \prod_{i=1}^{n} \Gamma_i$ represents the basis containing an ordered set occurrences, one from each of the n spaces.

Axiom 4: (Product Axiom) Let $(\Gamma_k, \mathcal{L}_k, \mathcal{M}_k)$ be uncertainty spaces for $k = 1, 2, \ldots$ The product uncertain measure $\mathcal{M}$ is an uncertain measure satisfying

$$\mathcal{M}\left\{\prod_{k=1}^{\infty} \Lambda_k\right\} = \bigwedge_{k=1}^{\infty} \mathcal{M}_k\{\Lambda_k\}$$

where Λ_k are arbitrary chosen events from $\mathcal{L}_k$ for $k = 1, 2, \ldots$, respectively. In the next axiom, the $\bigwedge$ represents the

minimum operator. This axiom says that the belief level in an ordered set of events is the minimum of the beliefs of those events.

How probabilistic measure is different from uncertainty measure? We take an example to establish how probability measures are different than the uncertainty measure. Assume that we do not know whether service B depends on service A or not. So we assume that the belief that B depends on A is denoted as η which is equal to 0.5. Suppose we want to find out the belief that out of 100 attempts, service B will be always called by service A. Using the probability theory (assuming that B depends on A with a probability of ξ = 0.5) we will find that this event happens with a probability of $\prod_{k=1}^{100} \xi = (0.5)^{100} = 7.8 \times 10^{-31} \approx 0$, i.e. this event happens with a negligible probability. On the other hand using the Axiom 4 of uncertainty theory reveals that event happens with a certainty of $\bigwedge_{k=1}^{100} \eta = 0.5$. Thus probability theory and uncertainty theory results in contradictory results. Notice that if the dependency exists invoking A will always call service B, if not then B will never be called. This phenomenon cannot be modelled by the probability theory as it is not based on frequency of outcomes.

III. CloudMiner: Overview and Model

We represent the enterprise cloud network as a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_i : i = 1, \ldots, n_\mathcal{V}\}$ denotes a set of vertices which are routers and hosts, and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ denotes a set of edges that interconnect the vertices $\mathcal{V}$. To represent the service dependencies we define $G_D = (V_S, A)$, where $V_S \subset \mathcal{V}$ denote the service vertices and $A = \{a_{ij} : i, j \in V_S\}$ denotes the dependency between services. Let $C = \mathcal{V} \cup \mathcal{E}$ denotes the set of all components for failure diagnosis.

We assume that the network topology is configured with dynamic routing (i.e., Open Shortest Path First (OSPF) or Routing Information Protocol (RIP)) which usually selects and installs the shortest paths to all destinations. The dynamic routing automatically reconverges and selects the next shortest path if the shortest path failed. Also, we assume that the source routing, where the sender can partially or wholly specify the route the flow can take to the destination, is not allowed.

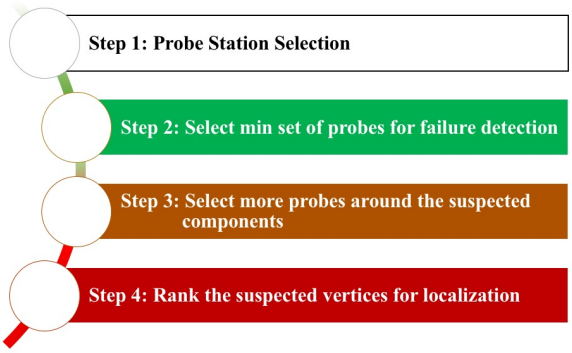


Fig. 3. Diagnosis Approach for CloudMiner

Fig. 3 shows the outline of our proposed CloudMiner framework. We first discuss our approach for placing the minimum number of probe stations assuming the network connectivity is known. We next propose our approach for finding the minimum set of probes for detecting the faulty components with some level of certainty. CloudMiner generates these set of probes in some periodic intervals. If one or more probes fail, then CloudMiner generates more probes aggressively around the suspected components, for diagnosing the faulty components. Finally CloudMiner generates a score for the components that are likely to be faulty. The detailed stages are summarized in the following subsections. Table II depicts the necessary notations.

TABLE II. Table of Notations

C	$\triangleq$	Set of components
P	$\triangleq$	Set of potential probing stations
$\mathbb{T}$	$\triangleq$	Set of all probes
n_T	$\triangleq$	Total number of probes
ρ_i	$\triangleq$	Cost (or length) of running probe i
$x_i \in 0, 1$	$\triangleq$	Whether or not probe i is chosen
t_i	$\triangleq$	i-th probe
b_i^c	$\triangleq$	Uncertain measure of detecting a failure at component c by probe t_i
$LOS(c)$	$\triangleq$	Level of suspiciousness of component c
$\mathbb{P}(c)/\mathbb{F}(c)$	$\triangleq$	Number of passed and failed test cases corresponding to the component c
$\mathbb{T}_P/\mathbb{T}_F$	$\triangleq$	Total number of test cases that pass/fail

A. Different types of probes

There are two types of probes; network probes and service probes. A network probe tests the reachability of the destination router or host using ping or trace-route probe. The ping probe passes if and only if there is at least one path to the destination and the destination is alive and working properly. The trace-route probe provides the status of every hop in the path to the destination, and thus it checks a specific way of reaching the destination, rather than any working path. The trace-route probe is a slow probe because it checks the status of each hop along the path and cannot be used to probe services and their dependencies.

A service probe tests the reachability of the destination service similarly, but it uses a service probe (i.e., HTTP, FTP, SMTP). Note that in addition to the path/node issues, a service probe would fail if any of its direct dependencies is not working correctly. For example, if the service depends on a database, it will fail if the database did not respond. A service can also be tested by a probe that does not interact with its direct dependencies. For example, a probe could be sent to test the service up/down without interacting with the database.

B. Probe station placement

For full diagnosing capability, we need to have a sufficient number of probing stations in appropriate locations in the network to ensure that failure at any component can be detected. Fig. 4 shows the limitations of using a small number of testing stations. Assume that we just install a testing station at node 1. Now the probing station at node 1 can send a ping probe to check whether node 2 is alive. On the other hand by sending a trace-route to node 2 it can find out whether the edge $(1, 2)$ is working properly (if the link $(1, 2)$ is down then the trace-route will take the next shortest path $1 \rightarrow 3 \rightarrow 2$ to

reach node 2). Similarly the probe station can detect whether the edges $(1,3)$ and $(2,5)$ are alive.

However a probe station at node 1 cannot guarantee whether failure at edges $(3,4)$ and $(2,4)$ will be detected. This is because there are two shortest paths (with equal costs) between node 1 and node 4 (through nodes 2 and 3). Thus analyzing a trace-route through a path does not reveal whether the edges in the other path are alive. Due to this ambiguity, detecting failures at edges $(2,3),(2,4),(3,4),(4,5)$ are not guaranteed by installing a testing station at node 1. Installing another station at node 3 breaks this ambiguity for edges $(2,3),(3,4)$, as shown in Fig. 4(b). However it still cannot resolve the issue for edges $(2,4),(4,5)$. Finally Fig. 4(c) shows that by installing probing stations at nodes $1,3,4$, we can indeed achieve full diagnosis capability. In the following we propose a scheme for installing the probing stations at few strategic locations.

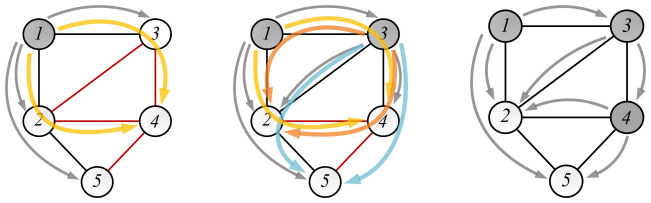


Fig. 4. Edge coverage with different number of probing stations

Let P denotes the set of all potential probing stations locations. Given the potential locations we select the minimum number of probing stations, $P' \subseteq P$, such that we can detect failures at all components in the network. We model this problem as a minimum set cover problem, which is a well known NP-hard problem [18]. The components that are detected by probing any station $p \in P$ are kept in a subset; we define that *p covers the components in that subset*. Let $\delta(p)$ denotes the subset of components that are covered by the probing station p. $\delta(p)$ can be computed by finding the shortest paths from p to each vertex in the graph $\mathcal{G}$. If there are multiple shortest paths of equal cost, then p can reliably cover the components *that fall in the intersection of these paths*. We thus add the components in the intersection to $\delta(p)$. In case there exists a single shortest path, we add the components in the shortest path to $\delta(p)$.

Now given the set of components C and the subsets $\delta(p)$ corresponding to each $p \in P$, the probing station placement problem is to find a collection of these subsets that covers all the components. The problem can be solved efficiently using the standard greedy heuristics for the set cover problem, depicted in Algorithm 1. Algorithm 1 starts with an empty set P' (line 1) and picks the probing station that covers the maximum number of uncovered components at each iteration (line 2-5). The process goes on until all the components are covered. The greedy algorithm in Algorithm 1 is a $(\ln|C|+1)$-approximation of the optimal set cover problem [19].

C. Select probes for failure detection

After selecting the probing stations, we generate the set of all possible probes $\mathbb{T} = \{t_i : 1 \leq i \leq n_T\}$ from those probing stations. We generate the probes based on the knowledge of $\mathcal{G}$ and G_D, which we assume remains fixed (with some of the dependencies as uncertain) during the period of the testing.

Algorithm 1 Installing the Probing Stations

Input: Set of components C, collection $\delta(P)$;
Output: P';
1: $P' = \emptyset$;
2: **while** $|\delta(P')| < |C|$) **do**
3: Select a $p \in P$ that maximizes $|\delta(P' \cup p) - \delta(P')|$;
4: $P' = P' \cup p$;
5: **end while**
6: **return** P';

As mentioned earlier, although shortest paths are chosen by the probes in the normal operation, in case of failures the probes take other paths to reach the destination. Consider the topology in Fig. 5 with shortest path routing. We have two services s_1 (web-server) and s_2 (database) with dependency $s_1 \rightarrow s_2$, and three probing stations p_1, p_2 and p_3. A HTTP probe (p_1, s_1) would not detect the failure of the edge $(1,2)$, because the routers will find another route through node 3. Even if both edges $(1,2)$ and $(3,2)$ fail, the HTTP probe (p_1, s_1) would find a route through 4 and would still pass. However, if we are certain about the existence of the dependency $s_1 \rightarrow s_2$, then the HTTP probe (p_1, s_1) would detect any failure to any of the components $\{1, 2, 5, s_1, h_1, s_2, h_2\}$ even if certain edges fail. For example, in Fig. 5 with $k = 3$, the k-shortest paths of the probe (p_1, s_1)=$\{\langle$ 1, $(1,2)$, 2, $(2, h_1)$, (s_1, h_1), s_1, h_1 $\rangle$, $\langle$ 1, $(1,3)$, 3, $(3,2)$, 2, $(2, h_1)$, s_1, h_1 $\rangle$ $\langle$ 1, $(1,3)$, 3, $(3,4)$, 4, $(4,2)$, 2, $(2, h_1)$, (s_1, h_1), s_1, h_1 $\rangle$ $\}$. We notice that the intersection of the three probes is $\{1, 2, (2, h_1), (s_1, h_1), s_1, h_1\}$, thus, we are confident that the probe (p_1, s_1) will always detect any failure in these components (with some certainty) in the intersection.

On the other hand, a trace-route probe (p_1, s_1) will detect a failure at the edge $(1,2)$ as mentioned in section III-B. However, trace-route probe $(1,4)$ cannot guarantee detecting failures in the edges $(3,4)$ and $(2,4)$, due to the presence of multiple equal-cost shortest paths.

In light of the above, we first generate the k-shortest paths[2] from each probing station $p \in P'$ to all other vertices (routers, hosts, and services) in the graph $\mathcal{G}$. For any probe between any probe station and network vertices, *the components that belong to the intersection of k-shortest paths can be detected with some level of certainty*. In case of trace-route probes, a probe can detect components that are on the unique shortest path. Otherwise in case of multiple shortest paths (with equal cost), the trace-route probe detects failures at the components that fall in the intersection of these paths. The uncertainty measures corresponding to the components are assigned based on the inter-dependencies between the services. Networks, data centers, and services configuration usually follow regulations and industry best practices. *We will assume that the certainty is derived based on these best practices*. In addition to that the inter-dependency information in between the services can also be derived using tools like NSD-Miner [2], [3]. However, the detailed procedure of finding the uncertainty measures are beyond the scope of this paper. The network probes are constructed from each probing station to all non-service vertices, whereas the service probes are constructed from the probing stations to all services.

[2] k can be any number

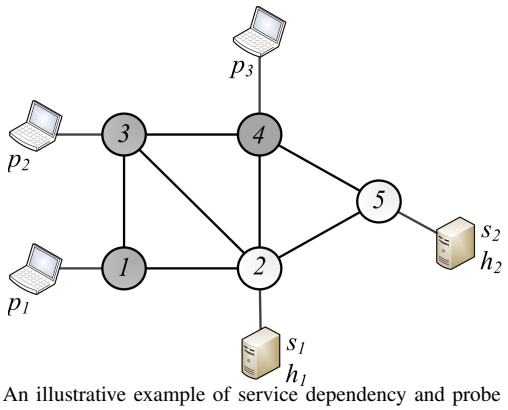


Fig. 5. An illustrative example of service dependency and probe selection

Next we select the minimum set of probes for network monitoring and failure detection such that if any component $c \in C$ failed, at least one probe will fail. Suppose that we have a total of n_T probes, and x_i is a binary variable which is 1 if probe t_i is selected and 0 otherwise. Also assume that b_i^c is the uncertainty measure of detecting a failure at component c by probe t_i. Then the total certainty is simply the maximum over all the probes selected, which can be expressed as:

$$R_c = b_1^c x_1 \bigvee b_2^c x_2 \bigvee b_3^c x_3 \ldots \bigvee b_{n_T}^c x_{n_T} = \bigvee_{i=1}^{n_T} b_i^c x_i \quad (1)$$

We assign each probe t_i with a *price* function ρ_i which depends on its length, as well as the type of the probe (i.e. HTTP probe or trace-route probe). Running trace-route probes result in higher cost/overhead than the HTTP probes. With this we define the following optimization problem which selects the set of probes that minimizes the cost of running them, while ensures that the reliability index for detecting any component is more than a predefined threshold α_1.

$$\boxed{\text{Min} \sum_{i=1}^{n_T} \rho_i . x_i \qquad \text{subject to (\textbf{C1}): } R_c \geq \alpha_1 \ \ \forall c \in C} \quad (2)$$

For ρ_i = 1 and $b_i^c = \alpha_1 = 1$, Problem 2 becomes a minimum set cover problem which is NP-hard. We thus propose the following heuristic. Let $\chi(T)$ denote the total uncertainty measure of detecting failures across all the components for a probe set T, i.e.,

$$\chi(T) = \sum_{c \in C} \bigvee_{t_i \in T} b_i^c \quad (3)$$

Then we can devise a greedy heuristic shown in Algorithm 2. Here $\mathbb{T}$ is the set of all possible probes that the test stations can run. Initially we start with an empty set T (line 1) and iteratively add the probes in T which minimize the cost of running them divided by the reliability gain (line 2-4).

Lemma 1: With $b_i^c = \alpha_1 = 1$, problem 2 becomes a minimum weighted set-cover problem. In that case Algorithm 2 returns a set cover of weight at most $\frac{1}{\Upsilon}$ times that of the optimal, where Υ is the size of the largest subset size.

Proof: When $b_i^c = 1$, $\chi(T \cup t_i) - \chi(T)$ generates the number of components in t_i that are not yet covered by T. Thus Algorithm 2 becomes the greedy solution for the minimum set cover problem as mentioned in [20], which is proven to be at most $\frac{1}{\Upsilon}$ times that of the optimal. ■

Algorithm 2 Minimum Probe Selection

Input: Set of components C, Probe set $\mathbb{T}$;
Output: Selected set of probes T;
1: $T = \emptyset$
2: **while** Constraint (**C1**) is not satisfied **do**
3: Select a probe $t_i \in \mathbb{T}$ that minimizes the price per reliability gain, i.e. $\frac{\rho_i}{\chi(T \cup t_i) - \chi(T)}$;
4: $T = T \cup t_i$;
5: **end while**
6: **return** T;

D. Select more probes aggressively for diagnosing the failures

The selected probes are run periodically; the *diagnosis phase* starts whenever one or more probes fail. The components which are covered by the failed probe(s) are marked as "suspected" components. Assume that ζ is the set of the suspected components, and $\mathcal{S}$ is the set of probes which covers at least one of suspected components with some certainty. We next run more probes aggressively that are targeted only for the components in ζ only. We define the following reliability index Γ_c corresponding to component c which has *stricter* requirement than that in equation(1):

$$\Gamma_c = \eta\text{-}max \left(b_1^c x_1, b_2^c x_2, b_3^c x_3 \ldots b_{|\mathcal{S}|}^c x_{|\mathcal{S}|} \right) \quad (4)$$

With this we define the following optimization problem which selects the minimum set of probes that ensure that failure at each component in ζ is detected by at least η probes with an uncertainty measure more than α_2.

$$\boxed{\text{Min} \sum_{t_i \in \mathcal{S}} x_i \qquad \text{subject to (\textbf{C2}): } \Gamma_c \geq \alpha_2 \ \ \forall c \in \zeta} \quad (5)$$

The approximation algorithm corresponding to Problem 5 is identical to Algorithm 2 and thus is ignored for brevity.

E. Ranking of suspected nodes

The pass/fail information of the probes from different probe stations along with the components that are executed by each test cases, are next used to narrow down the problematic components. Due to the lack of accurate and ambiguous dependency information, and because of the fact that a test can fail due to any problem somewhere down the dependency chain, accurately finding out the problematic components are non-trivial. We thus use a metric named *likelihood of suspiciousness (LOS)* of a components c being problematic as follows:

$$LOS(c) = \frac{\%\text{passed}(c)}{\%\text{passed}(c) + \%\text{failed}(c)} = \frac{\mathbb{F}(c)/\mathbb{T}_F}{\mathbb{P}(c)/\mathbb{T}_P + \mathbb{F}(c)/\mathbb{T}_F} \quad (6)$$

where $\mathbb{P}(c)$ and $\mathbb{F}(c)$ are the number of passed and failed test cases corresponding to the component c respectively, and $\mathbb{T}_P$ and $\mathbb{T}_F$ are the total number of test cases that pass and fail respectively. The intuition behind the LOS calculation is that the components that are executed primarily by failed test cases are highly suspicious as being faulty, whereas the components that are executed primarily by passed test cases are not likely to be faulty. Notice that due to uncertainties in calculating dependencies and ambiguities, a passed probe

does not always mean that all the components under that probe is well-configured. That is where the relevance of likelihood arises. Such matrices are extensively used and tested in localizing faults in software [15], [21], [22], however, their effectiveness in locating the failures/misconfigurations in enterprise environment is quite new.

Thus the LOS specifies a *ranking* of components that are under scanner of suspicion. Using the LOS score, the set of components that have the highest suspiciousness values are first considered by the operator for checking the failures. If the highest LOS component is configured properly, the remaining components should be examined in the sorted order of the decreasing suspiciousness values.

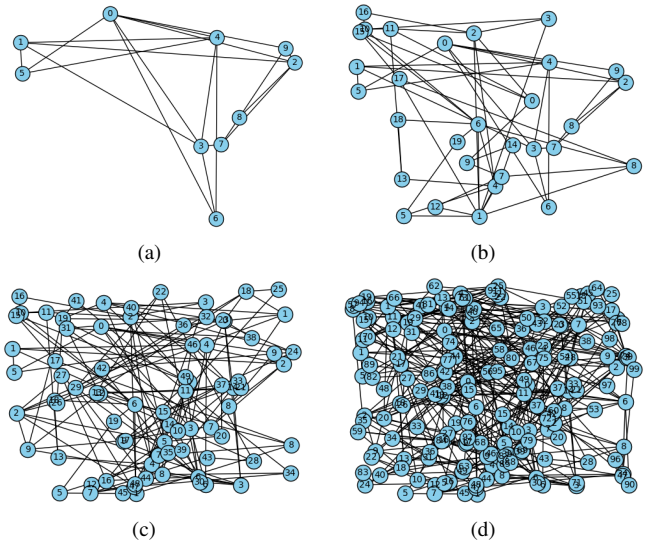
TABLE III. SUSPICIOUSNESS SCORE FOR THE TOPOLOGY IN FIG. 5

Components	Probes							LOS
	1	2	3	4	5	6	7	
$(2,4)$						•		0.00
1	•		•	•				0.00
2	•		•		•	•		0.67
3		•		•	•			0.75
4		•				•	•	0.00
5	•		•				•	0.00
h_1	•							0.00
$(h_2,5)$	•							0.00
$(1,2)$			•					0.00
$(2,h_1)$	•							0.00
s_2	•							0.00
h_2	•							0.00
(s_1,h_1)	•							0.00
(s_2,h_2)	•							0.00
$(3,4)$		•						0.00
$(2,3)$					•			1.00
$(2,5)$			•					0.00
$(1,3)$				•				0.00
s_1	•							0.00
$(4,5)$							•	0.00
Pass (P)/Fail(F)	P	P	P	P	F	P	P	

Table III illustrates an example for depicting the suspiciousness, using the network topology of Fig. 5. Assume that there are seven probes for monitoring all components. In Table III (•) denotes the components that are covered by the corresponding probes. One component is randomly selected as failed. As a result, one probe fails. The LOS scores are calculated accordingly that help localizing the failure. The table shows that the components that are not suspicious has 0 score and do not need fixing. The components with higher LOS are most likely failed components. Whereas, the components that are less scores have passed some probes and therefore are likely to be okay.

IV. EXPERIMENTAL EVALUATION

Simulation environment: To evaluate the performance of `CloudMiner` we used BRITE [23] to randomly generate *network* topologies with 10, 20, 50, and 100 routers as shown in Fig. 6. In BRITE generated networks, the number of edges is double the number of nodes. We generate additional nodes for services and hosts and connect them to the network. In this evaluation, we chose to add 40% of the number of routers as hosts. Each host runs one service. We randomly connect the hosts to one of the routers and each host connects to its service. We also generate a directed graph to represent the services' dependencies with random weights in the range (0, 1) on the arcs to represent the uncertainty of the dependencies. For example, the network with 10 routers has 20 edges to interconnect the routers, 4 services, 4 hosts, 4 edges to connect



(a) (b) (c) (d)

Fig. 6. Illustration of the topologies with (a) 10, (b) 20, (c) 50, and (d) 100 routers used in the simulations.

services to hosts, and 4 edges to connect hosts to a network router. Table IV summarizes the configuration of the evaluation cases, where $|N|$ and $|E|$ denote the number or routers and edges of the BRITE topology, $|H|$ denotes the number of hosts, $|S|$ denotes the number or services, $|(s,h)|$ and $|(n,h)|$ denote the number of service-to-host edges and host-to-network edges, and $|C|$ denotes the number of components. The price function of the probes are assumed to be equal to their length for simplicity.

TABLE IV. EVALUATION CASES

Case	$\|N\|$	$\|E\|$	$\|H\|$	$\|S\|$	$\|(s,h)\|$	$\|(n,h)\|$	$\|C\|$
Topo-1	10	20	4	4	4	4	46
Topo-2	20	40	8	8	8	8	92
Topo-3	50	100	20	20	20	20	230
Topo-4	100	200	40	40	40	40	460

Number of probing stations: We assume that we can connect a probing station to each BRITE network router. Thus, we start the evaluation with $|P| = 10, 20, 50, 100$. Fig. 7 shows the sizes of the initial set of probing stations and the minimum set of probing stations that are capable to detect failures in any component in the generated networks. From this figure we can observe that by using intelligent probe station placement, we can reduce the number of probe stations by $\sim$2-17 times.

Number of probes: After selecting the minimum number of probing stations, we ran Algorithm 2 assuming α_1 = 0.8 to select the minimum number of probes for failure detection. We compare the number of probes for the minimum and the maximum number (i.e. probing stations at all possible locations) of probing stations. Obviously, using more probing stations reduces the number of probes, as shown in Fig. 8. However, the gain in number of probes may not be worthwhile. For example in case of topology-4, an increase in number of probing stations from 6 to 100 reduces the number of probes by $\sim$27%.

Fig. 9 shows the number of probes for $\alpha_1 = 0.8$ 1.0, i.e., for partial and full knowledge regarding the inter-dependencies. Obviously, the number of probes is lower in the latter case because of exploiting the complete knowledge of dependencies.

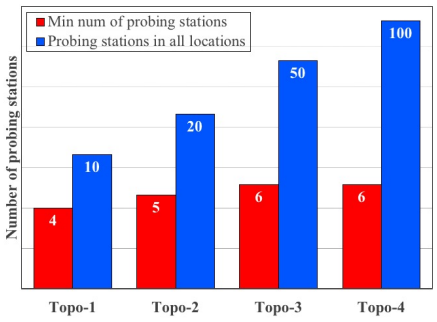


Fig. 7. Min vs. all probing stations

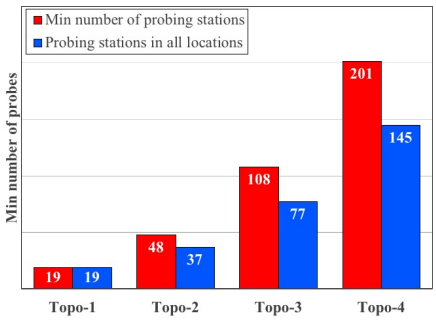


Fig. 8. Min probes with min vs. all probing stations

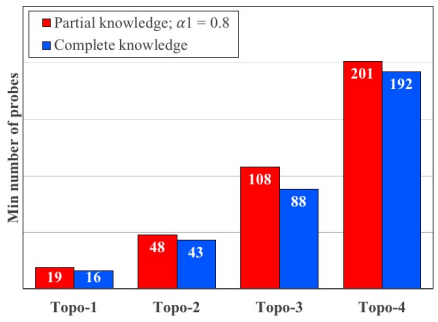


Fig. 9. Min num of probes with different certainty

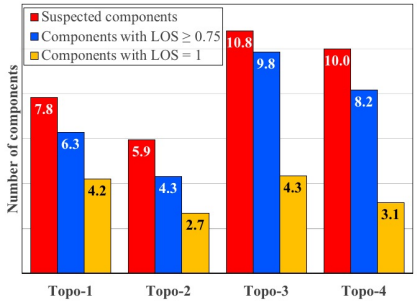


Fig. 10. Number of components within ζ, with LOS score above 0.75, and with LOS equal to 1.

Number of suspected components: Next we introduce a random fault within the network components and computed their LOS scores assuming $\alpha_2 = 0.8$. We have considered single-fault scenario for simplicity. We repeated this process 50 times and recorded the suspected components. Fig. 10 shows the comparison of the number of suspected components in ζ, the number of components whose LOS score is more than 0.75, and the number of components with LOS equal to 1. From Fig. 10 we can observe that the diagnosis phase reduces the number of components further by $\sim$2-3 times. We have also observed that the failed component is always in the suspected components and has an LOS equal to 1. We can also observe that the number of components with LOS equal to 1 is as low as $\sim$3 for topology-4 that consists of a total of 460 components. This shows that `CloudMiner` can effectively diagnose the failed components in an enterprise cloud environment.

V. Conclusion

In this paper, we have addressed the problem of selecting the minimum number of probing stations in an enterprise cloud environment and studied its impact on selecting the minimum number of probes for failure detection and diagnosis. Also, we studied the impact of having partial knowledge of the inter-dependencies among the network services and how to model it using a novel uncertainty theory. Further, we defined the failure localization problem in the presence of uncertainties and proposed an efficient heuristic algorithm for solving it. In the future, we would like to emulate such diagnosis scenarios in Common Open Research Emulator (CORE) [24]. In this way we will implement and validate our detection and localization solutions in an cloud environment that is closer to real-life.

References

[1] Endo *et al.*, "Minimizing and managing cloud failures," *IEEE Computer*, vol. 50, no. 11, pp. 86–90, 2017.

[2] Natarajan *et al.*, "NSDMiner: Automated discovery of network service dependencies," in *IEEE INFOCOM*, 2012, pp. 2507–2515.

[3] Peddycord III *et al.*, "On the accurate identification of network service dependencies in distributed systems," in *LISA*, 2012.

[4] Liu, *Uncertainty Theory*, ser. Studies in Fuzziness and Soft Computing. Springer, 2007.

[5] Ozmutlu *et al.*, "Zone recovery methodology for probe-subset selection in end-to-end network monitoring," in *IEEE NOMS*, 2002, pp. 451–464.

[6] Zheng *et al.*, "Detecting and localizing large-scale router failures using active probes," in *IEEE MILCOM*, 2011, pp. 1170–1175.

[7] Rish *et al.*, "Real-time problem determination in distributed systems using active probing," in *IEEE NOMS*, vol. 1, 2004, pp. 133–146.

[8] Rish *et al.*, "Adaptive diagnosis in distributed systems," *IEEE Transactions on neural networks*, vol. 16, no. 5, pp. 1088–1109, 2005.

[9] Li and Thottan, "End-to-end service quality measurement using source-routed probes." in *INFOCOM*, 2006.

[10] Breitbart *et al.*, "Efficiently monitoring bandwidth and latency in ip networks," in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2. IEEE, 2001, pp. 933–942.

[11] Jeswani *et al.*, "Probe station selection algorithms for fault management in computer networks," in *Communication Systems and Networks (COMSNETS), 2010 Second International Conference on*. IEEE, 2010, pp. 1–9.

[12] Han *et al.*, "The maximum flow problem of uncertain network," *Information Sciences*, vol. 265, pp. 167–175, 2014.

[13] Zhang and Peng, "Uncertain programming model for uncertain optimal assignment problem," *Applied Mathematical Modelling*, vol. 37, no. 9, pp. 6458–6468, 2013.

[14] Chen *et al.*, "Uncertain programming model for uncertain minimum weight vertex covering problem," *Journal of Intelligent Manufacturing*, vol. 28, no. 3, pp. 625–632, 2017.

[15] Jones and Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ACM ASE*, 2005, pp. 273–282.

[16] Meng *et al.*, "It troubleshooting with drift analysis in the devops era," *IBM Journal of Research and Development*, vol. 61, no. 1, pp. 6:62–6:73, 2017.

[17] Chen *et al.*, "Automating network application dependency discovery: Experiences, limitations, and new solutions." in *OSDI*, vol. 8, 2008, pp. 117–130.

[18] Garey and Johnson, "A guide to the theory of np-completeness," *WH Freemann, New York*, 1979.

[19] Cormen *et al.*, *Introduction to algorithms*. MIT press, 2009.

[20] Young, "Greedy set-cover algorithms," URL https://link.springer.com/content/pdf/10.1007/978-3-642-27848-8_175-2.pdf, 2014.

[21] Bandyopadhyay and Ghosh, "On the effectiveness of the tarantula fault localization technique for different fault classes," in *IEEE HASE*, 2011, pp. 317–324.

[22] Jones *et al.*, "Visualization of test information to assist fault localization," in *ICSE*, 2002, pp. 467–477.

[23] Medina *et al.*, "Brite: A flexible generator of internet topologies," Boston, MA, USA, Tech. Rep., 2000.

[24] Ahrenholz *et al.*, "Core: A real-time network emulator," in *IEEE MILCOM*, 2008, pp. 1–7.