NetBouncer: Active Device and Link Failure Localization in Data Center Networks

Cheng Tan[1], Ze Jin[2], Chuanxiong Guo[3], Tianrong Zhang[4], Haitao Wu[5], Karl Deng[4], Dongming Bi[4], and Dong Xiang[4]

[1]New York University, [2]Cornell University, [3]Bytedance, [4]Microsoft, [5]Google

Abstract

The availability of data center services is jeopardized by various network incidents. One of the biggest challenges for network incident handling is to accurately localize the failures, among millions of servers and tens of thousands of network devices. In this paper, we propose NetBouncer, a failure localization system that leverages the IP-in-IP technique to actively probe paths in a data center network. NetBouncer provides a complete failure localization framework which is capable of detecting both device and link failures. It further introduces an algorithm for high accuracy link failure inference that is resilient to real-world data inconsistency by integrating both our troubleshooting domain knowledge and machine learning techniques. NetBouncer has been deployed in Microsoft Azure's data centers for three years. And in practice, it produced no false positives and only a few false negatives so far.

1 Introduction

As many critical services have been hosted on the cloud (e.g., search, IaaS-based VMs, and databases), enormous data centers have been built which contain millions of machines and tens of thousands of network devices. In such a large-scale data center network, failures and incidents are inevitable, including routing misconfigurations, link flaps, network device hardware failures, and network device software bugs [19, 20, 22, 23, 28, 45]. As the foundation of network troubleshooting, failure localization becomes essential for maintaining a highly available data center.

Localizing network failures in a large-scale data center is challenging. Given that nowadays data centers have highly duplicated paths between any two end-hosts, it is unclear to the end-hosts which links or devices should be blamed when a failure happens (e.g., TCP retransmits). And, because of the Equal-Cost Multi-Path (ECMP) routing protocol, even routers are unaware of the whole routing path of a packet.

Moreover, recent research [23, 27, 28] reveals that *gray failure*s, which are partial or subtle malfunctions, are prevalent within data centers. They cause the major availability breakdowns and performance anomalies in cloud environments [28]. Different from *fail-stop failure*s, gray failures drop packets probabilistically, and hence cannot be detected by simply evaluating connectivity.

In order to localize failures and be readily deployable in a *production* data center, a failure localization system needs to satisfy three key requirements, which previous systems [1, 16, 18, 23, 37, 44, 46] fail to meet simultaneously.

First, as for detecting gray failures, the failure localization system needs an end-host's perspective. Gray failures have been characterized as "differential observability" [28], meaning that the failures are perceived differently by end-hosts and other network entities (e.g., switches). Therefore, traditional monitoring systems, which query switches for packet loss (e.g., SNMP, NetFlow), are unable to observe gray failures.

Second, to be readily deployable in practice, the monitoring system should be compatible with commodity hardware, the existing software stack and networking protocols. Previous systems which need special hardware support [37], substantial modification on the hypervisor [40] or tweak standard bits on network packets [46] are unable to be readily deployed in production data centers.

Third, localizing failures should be precise and accurate, in terms of pinpointing failures in fine-granularity (i.e., towards links and devices) and incurring few false positives or negatives. Some prior systems, like Pingmesh [23] and NetNORAD [1], can only pinpoint failures in a region, which needs extra efforts to discover the actual errors. And others [16, 18, 44] incur numerous false positives and false negatives when exposed to gray failures and real-world data inconsistency.

In this paper, we introduce NetBouncer, an active probing system that detects device failures and infers link failures from end-to-end probing data. NetBouncer satisfies the previous requirements by actively sending probing packets from the servers, which doesn't need any modification in the network or underlying software stack. In order to localize failures accurately, NetBouncer provides a complete failure localization framework targeting data center networks, which incorporates real-world observations and troubleshooting domain knowledge. In particular, NetBouncer introduces:

- *An efficient and compatible path probing method* (§3). We design a probing method called *packet bouncing* to probe a designated path. It is built on top of the IP-in-IP technique [7, 47], which has been implemented in ASIC of modern commodity switches. Hence, packet bouncing is compatible to current data center networks and efficient without consuming switch CPU cycles.
- *A probing plan which is able to distinguish device failures and is proved to be link-identifiable* (§4). A probing plan is a set of paths which will be probed. Based on an observation that the vast majority of the network is

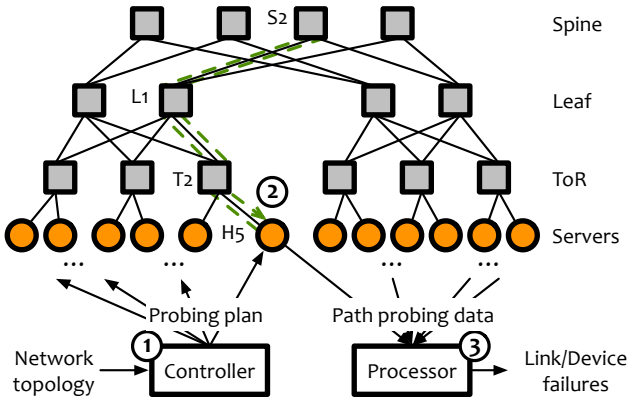


Figure 1: NetBouncer's workflow: ①, the controller designs a probing plan and sends it to all the probing servers. ②, the servers follow the plan to probe the paths in the network. ③, the processor collects the probing data and infers the faulty devices and links.

healthy, we conceive a probing plan which reveals the device failures. And, by separating the faulty devices from the network, we *prove* that the remaining network is link-identifiable, meaning that the status of each link can be uniquely identified from the end-to-end path probing.

- *A link failure inference algorithm against real-world data inconsistency* (§5). A link-identifiable probing is not sufficient for pinpointing failures due to real-world data inconsistency. We formulate an optimization problem which incorporates our troubleshooting domain knowledge, and thus is resilient to the data inconsistency. And, by leveraging the characteristic of the network, we propose an efficient algorithm to infer link failures by solving this optimization problem.

NetBouncer has been implemented and deployed (§8) in Microsoft Azure for three years, and it has detected many network failures overlooked by traditional monitoring systems.

2 NetBouncer overview

NetBouncer is an active probing system which infers the device and link failures from the path probing data. NetBouncer's workflow is depicted in Figure 1, which is divided into three phases as follows.

Probing plan design (① in Figure 1). NetBouncer has one central controller which produces a probing plan based on the network topology. A probing plan is a set of paths that would be probed within one probing epoch. Usually, the probing plan remains unchanged. Yet, for cases such as topology changes or probing frequency adjustments, the controller would update the probing plan.

An eligible probing plan should be link-identifiable, meaning that the probing paths should cover all links and more importantly, provide enough information to determine the status of every single link. However, the constraints in developing real-world systems make it challenging to design a proper probing plan (§4.2).

Based on an observation that the vast majority of the links in a network is healthy, we prove the *sufficient probing theorem* (§4.3) which guarantees that NetBouncer's probing plan is link-identifiable in a Clos network when at least one healthy path crosses each switch.

Efficient path probing via IP-in-IP (② in Figure 1). Based on the probing plan, the servers send probing packets through the network. The packet's routing path (e.g., $H_5 \rightarrow T_2 \rightarrow L_1 \rightarrow S_2 \rightarrow L_1 \rightarrow T_2 \rightarrow H_5$ in Figure 1) is designated using the IP-in-IP technique (§3.1). After each probing epoch, the servers upload their probing data (i.e., the number of packets sent and received on each path) to NetBouncer's processor.

NetBouncer needs a path probing scheme that can explicitly identify paths and imposes negligible overheads. Because the data center network is a performance-sensitive environment, even a small throughput decrease or latency increase can be a problem [30]. NetBouncer leverages the hardware feature in modern switches – the IP-in-IP [7, 47] technique – to explicitly probe paths with low cost.

Failure inference from path measurements (③ in Figure 1). The processor collects the probing data and runs an algorithm to infer the device and link failures (§4.4, §5.2). The results are then sent to the operators for further troubleshooting and failure processing.

The main challenge of inferring failures comes from the data inconsistency in the data center environment (§5.1). We've analyzed some real-world cases and encoded our troubleshooting domain knowledge into a specialized quadratic regularization term (§5.2). On top of that, we develop an efficient algorithm based on coordinate descent (CD) [53] which leverages the sparse characteristic of links in all paths. And the algorithm is more than one order of magnitude faster than off-the-shelf SGD solutions.

NetBouncer's targets and limitations. NetBouncer targets non-transient (no shorter than the interval between two probings), packet-loss network incidents. Though its expertise is on detecting gray failures which would be overlooked by traditional monitoring systems, any other packet-loss related incidents are also under its radar.

Admittedly, there are cases where NetBouncer fails to detect (see false negatives in §8). We discuss NetBouncer's limitations in more details in §9.

3 Path probing via packet bouncing

In a data center environment, the probing scheme of a troubleshooting system needs to satisfy two main requirements: first, the probing scheme should be able to pinpoint the routing path of probing packets, because a data center network provides many duplicated paths between two end-hosts.

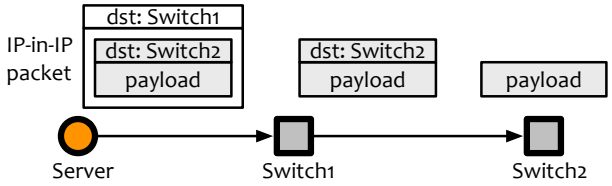
Second, the probing scheme should consume little network resources, in terms of switch CPUs and network bandwidth. This is especially important under heavy workloads when failures are more likely to happen.

Conventional probing tools fail to meet both requirements simultaneously. In short, ping-based probing is unable to pinpoint the routing path; Tracert consumes switch CPUs, which might adversely impact the reliability and manageability of the network.

NetBouncer designs and implements an approach called *packet bouncing*, which takes advantage of the IP-in-IP technique to accomplish both requirements. Other source routing schemes [15, 21, 26, 29] might also be plausible, but require much more deployment effort. NetBouncer uses probes from end-hosts. With programmable switches, it is possible to deploy probing agents at switches and probe each link individually. NetBouncer chooses end-host based approach as most of the switches in our data centers are still non-programmable. Nonetheless, NetBouncer's failure localization algorithm applies to switch-based approaches as well.

3.1 IP-in-IP basics

IP-in-IP [7, 47] is an IP tunneling protocol that encapsulates one IP packet in another IP packet. This protocol has been implemented in the modern commodity switches (in ASIC) which allows devising a specific probing path without involving the CPUs of switches.



NetBouncer utilizes this IP-in-IP technique to explicitly probe one path by encapsulating the desired destination in the nested IP packets. In the above abstract example, NetBouncer is able to probe a certain path (Server $\rightarrow$ Switch$_1$ $\rightarrow$ Switch$_2$) by encapsulating the final destination (Switch$_2$) in the inner IP header and the intermediate hop (Switch$_1$) in the outer IP header. The switch that receives the IP-in-IP packets (i.e., Switch$_1$) would decapsulate the outer IP header and forward the packet to its next destination.

Indeed, some legacy or low-end switches might not support IP-in-IP in hardware. We do consider this challenge and design NetBouncer as only requiring the top-layer switches (i.e., core switches) having such support (details in §4.3). We believe that the core switches in a modern data center would be high-end with such support.

3.2 Packet bouncing

On top of the IP-in-IP technique, NetBouncer adopts a path probing strategy called packet bouncing. Namely, the probing server chooses a switch as its destination and inquires the switch to bounce the packet back. As an example in Figure 1, a server (e.g., H_5) sends a probing packet to a switch (e.g., S_2). The probing path contains the route from the server to the switch ($H_5 \rightarrow T_2 \rightarrow L_1 \rightarrow S_2$) and its "bouncing back" route ($S_2 \rightarrow L_1 \rightarrow T_2 \rightarrow H_5$).

In NetBouncer's target network, the Clos network [2, 48], packet bouncing simplifies NetBouncer's model, design and implementation, due to the following three reasons.

(1) It minimizes the number of IP-in-IP headers NetBouncer has to prepare. The packet bouncing only needs to prepare one IP-in-IP header which leads to a simple implementation. Given a Clos network, *only one path* exists from a server to a upper-layer switch (also observed by [46]). Hence, preparing an IP-in-IP packet (to a upper-layer switch) only needs one outer IP header (with its destination as that switch), which remarkably simplifies the implementation.

(2) Links are evaluated bidirectionally which leads to a simpler model. When packet bouncing is in use, all the links are evaluated bidirectionally. This simplifies NetBouncer's model, allowing the network graph to be undirected (§ 4.1). Indeed, this bidirectional evaluation cannot differentiate which direction of a link is dropping packets. However, in practice, this is not an issue because a link is problematic whichever direction drops packets.

(3) The sender and receiver are on the same server, which makes NetBouncer robust against server failures. Because of bouncing, the probing data for a certain path is preserved by one server, which is both the sender and the receiver. Thus, the probing data are "all or nothing". Otherwise, if the senders and receivers are different servers, NetBouncer has to consider the failures of senders (fewer sent packets, causing false negatives) or receivers (fewer received packets, causing false positives) or both, which makes the failure handling more complicated, especially in a large-scale distributed system.

4 Probing plan and device failure detection

This section proposes NetBouncer's failure localization model (§4.1) and introduces the challenges of probing path selection (§4.2) which motivates the probing plan design (§4.3) and device failure detection algorithm (§4.4).

We assume in this section that the success probability for each link is stable (i.e., remain the same among different measurements) which will be relaxed in the next section (§5).

4.1 Underlying model

We define a data center network as an undirected graph whose vertices are devices (e.g., servers, switches and routers) and edges are physical links. Each link has a *success probability*, which is denoted by x_i for the i^{th} link (link$_i$).

A path is a finite sequence of links which connect a sequence of devices. In NetBouncer, a probing path is

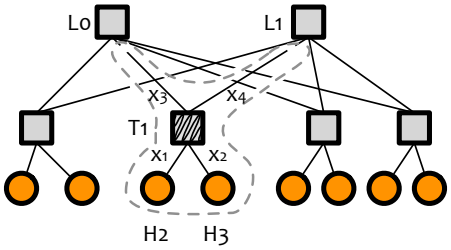


Figure 2: An unsolvable example. Switch T_1 cannot bounce packets. And, x_1, x_2, x_3, x_4 represent the success probabilities of link H_2-T_1, H_3-T_1, T_1-L_0 and T_1-L_1 respectively.

the sequence of links traversed by a probing packet from its sender to receiver. A *path success probability* is the probability of successfully delivering one packet through all links within this path. We use y_j to indicate the success probability of the j^{th} path (path_j).

NetBouncer's model *assumes* that dropping packets on different links are independent events, which has been justified in earlier work [17, 18, 42] (also see §9 for more discussion). Thus, the probability of successfully delivering one packet through a path can be described as

$$y_j = \prod_{i:\ \text{link}_i \in \text{path}_j} x_i, \forall j, \tag{1}$$

where the success probability of path_j is the product of its link success probabilities.

In the context of failure localization, the path success probabilities (y_js) can be measured by sending probing packets through the paths, and our ultimate goal is to pinpoint the faulty links (whose success probabilities x_is are below a certain threshold) and faulty devices (whose associated links are faulty).

4.2 Real-world challenges for path selection

In order to localize failures, the first question we need to answer is: *which paths should be probed so that all the links are identifiable?* This link identifiability problem can be formalized as follows.

Given a network graph G and all its possible paths U, how to construct a set $A \subseteq U$, so that the set of equations $\{y_j = \prod_{\text{link}_i \in \text{path}_j} x_i \mid \text{path}_j \in A\}$ has a unique solution for all x_is.

Whether the above equations have a unique solution has been well-studied in the literature of linear algebra (by taking logarithm at both sides of Equation 1, it becomes linear). However, in reality, not all paths can be probed. The probing path must start and end at servers, since most switches cannot issue packets (most of the switches are non-programmable). Moreover, the bouncing scheme further restricts the sender and receiver to be the same server (§3.2).

Under such constraints, we notice that if *any* switch cannot "bounce" packets (i.e., doesn't support IP-in-IP), there is no unique solution. As an example, Figure 2 depicts a simple two-tier Clos network with switch T_1 (the shaded switch) unable to bounce packets. As a result, there doesn't exist a unique solution in the circled subgraph, which is illustrated as follows.

Suppose we probe all the possible paths in the circled subgraph (i.e., H_2-T_1-L_0, H_2-T_1-L_1, H_3-T_1-L_0 and H_3-T_1-L_1) and obtain four equations as

$$y_{\{H_2\text{-}T_1\text{-}L_0\}} = x_1 \times x_3, \quad y_{\{H_2\text{-}T_1\text{-}L_1\}} = x_1 \times x_4,$$
$$y_{\{H_3\text{-}T_1\text{-}L_0\}} = x_2 \times x_3, \quad y_{\{H_3\text{-}T_1\text{-}L_1\}} = x_2 \times x_4.$$

Intuitively, since one of the four equations is redundant ($y_{\{H_2\text{-}T_1\text{-}L_0\}} \times y_{\{H_3\text{-}T_1\text{-}L_1\}} = y_{\{H_2\text{-}T_1\text{-}L_1\}} \times y_{H\{H_3\text{-}T_1\text{-}L_0\}}$), the number of effective equations is smaller than the number of variables. Thus, there doesn't exist a unique solution *in general*.

Unfortunately, cases similar to the above example occur in a data center network for many reasons. On the one hand, some switches (especially ToR switches) may not support the IP-in-IP forwarding, so that they cannot bounce packets; On the other hand, delayed uploading and failures are common in a large-scale system. Within one epoch, the probing data from a certain switch may fail to be uploaded. More importantly, bouncing every single switch is expensive and thus not favorable in terms of the huge number of probing paths.

4.3 Link-identifiable probing plan

In view of the challenges when choosing the paths, finding a probing plan that has a unique solution is generally difficult. However, in the real-world scenario, we observe that the vast majority of the links in a network are well-behaved and thus most of the paths are healthy.

Motivated by this observation, we come up with the *sufficient probing theorem*, which proves that when the network is healthy (at least one healthy path passes each switch), a simple probing plan (probing all paths from the servers to the top-layer switches) is link-identifiable. By link-identifiable, we mean that this probing plan can guarantee a *unique* solution (i.e., a set of x_is) to the path selection problem (§4.2) which is consistent with our measurements (i.e., all the y_js). Therefore, this plan is used as NetBouncer's probing plan.

Theorem 1. *(sufficient probing theorem). In a Clos network with k layers of switches ($k \geq 1$), by probing all paths from the servers to the top-layer switches, we can uniquely infer the link success probabilities from the measured path success probabilities, if and only if at least one path with success probability 1 passes each switch.*

The intuition behind the proof of this theorem (see full version proof in appendix §A) is that if the success probability of a path is 1, all the links included by this path should also have success probabilities 1, considering the constraint $x_i \in [0,1], \forall i$.

Furthermore, from the proof, we can see that this theorem can be easily extended to all the *layered* networks. In fact, the Clos network is a special case of a general layered network, where switches on layer n only connect to switches on layers $n-1$ and $n+1$, switches on the same layer do not connect to each other, and servers connect only to the first-layer switches.

Most of the probing plan designs in the literature [11, 12, 36, 39, 44, 54] target how to minimize the number of probing paths. Reducing the probing path number, however, is not a goal of NetBouncer. In fact, redundant paths through one link can be considered as validations to each other. These validations in turn increase NetBouncer's accuracy.

4.4 Device failure detection

Using NetBouncer's probing plan, Theorem 1 provides a sufficient and necessary condition (i.e., *at least one path with success probability 1 passes each switch*) for the existence of a unique solution. By checking whether the above condition holds for each switch, we can split a Clos network into a solvable part (having a unique solution) and an unsolvable part (no unique solution).

The unsolvable part would be a collection of switches which fail to have even one good path across it. Since NetBouncer probes many paths (usually hundreds to thousands) across each switch, one switch is highly suspicious if it doesn't even have one good path through it. Hence, NetBouncer reports these switches as faulty devices to the operators.

Theoretically, the reported device can be a false positive if it happens to be surrounded by bad devices. However, this case is extremely rare since one switch usually connects to many devices. Thus, we are highly confident that the reported devices are faulty.

To sum up, the servers first probe paths based on the probing plan in §4.3. Then the processor collects all the probing data from the servers, and extracts the faulty devices (unsolvable part) from the network. Based on Theorem 1, the remaining subgraph has a unique solution for each link's success probability. Next the processor runs the link failure inference algorithm described in the next section (§5.2), and infers the faulty links. Finally, the processor reports the faulty devices and links to the operators. The algorithm running on NetBouncer's processor is depicted in Figure 3.

5 Link failure inference

The previous section describes NetBouncer's probing plan and algorithm for localizing device failures. Yet, the last jigsaw piece of NetBouncer's algorithm (Figure 3, line 6) is still missing: *how can one infer the link probabilities x_is from the end-to-end path measurements y_js?*

Define:
devs: all devices
$Y : path \to [0,1]$ // a map from a path to its success probability

```
 1: procedure PROCESSOR( )
 2:     (1) Collect probing data from agents as Y
 3:     (2) badDev ← DETECTBADDEVICES(Y) // line 9
 4:     // eliminate the unsolvable subgraph
 5:     (3) Y ← Y \ {path_r | path_r passes any device in badDev}
 6:     (4) badLink ← DETECTBADLINKS(Y) // in Figure 5, §5.2
 7:     return badDev,badLink
 8:
 9: procedure DETECTBADDEVICES(Y)
10:     badDev ← {}
11:     for dev_p in devs :
12:         goodPath ← False
13:         for all path_q passes dev_p :
14:             If Y[path_q] = 1 then goodPath ← True; break
15:         If not goodPath then badDev += dev_p
16:     return badDev
```

Figure 3: Algorithm running on NetBouncer processor.

In practice, the above inference problem cannot be resolved simply using linear algebra or least squares, because of the real-world data inconsistency.

5.1 Data inconsistency

In the real-world data center environment, the measurement data are usually inconsistent or even conflicting. Such data inconsistency derives from two main reasons:

- *Imperfect measurement.* The data center network is huge and its state changes constantly. Due to its gigantic size, all the paths cannot be probed simultaneously. Thus, different path probings may reflect different instantaneous states of the network. Moreover, as the probing sample size is limited (hundreds of packets per path), the measurements on each path are coarse-grained.
- *Accidental packet loss.* In a large-scale network, accidental errors are inevitable, which can happen on any layer (e.g., hypervisor, OS, TCP/IP library) of the execution stack as a result of bugs, misconfigurations, and failures.

These two reasons lead to inconsistency in the path probing data and further to misreporting (mostly false positives, reporting a well-behaved link as a faulty one). The reason why *accidental packet loss* introduces false positives is straightforward. As it incurs dropping packets which no link or device should be responsible for, such packets might be attributed to the non-lossy links which produces false positives.

As for the *imperfect measurement*, the reason why it causes false positives is that the inference results might *overfit* the imperfect measurements. We demonstrate this problem by a real-world example (Figure 4).

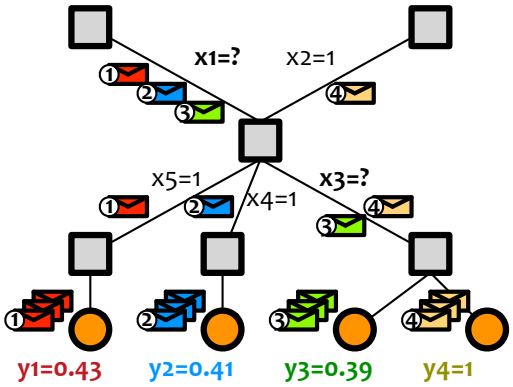


Figure 4: A false positive example of the least square solution overfitting imperfect measurement data. The circled number on the probing packets indicates which path this packet passes.

In Figure 4, we have priori knowledge that some links are good ($x_2 = x_4 = x_5 = 1$) , and we want to infer the link success probabilities x_1 and x_3 from observed path success probabilities ($y_1 = 0.43$, $y_2 = 0.41$, $y_3 = 0.39$, and $y_4 = 1$). Using the least squares approach we obtain the estimates $x_1 = 0.406$ and $x_3 = 0.965$, which indicates that both links are dropping packets. However, the faulty link with respect to x_3, unfortunately, is a false positive. Such false positive is caused by the imperfect measurements of y_1, y_2, y_3 as their observed success probabilities are slightly different. In this case, the least square results overfit the imperfect data when minimizing the fitting error.

To mitigate the above false positives, we introduce a specialized regularization term and propose a regularized estimator for the latent factor model to resolve the failure localization problem, which is described in the next section.

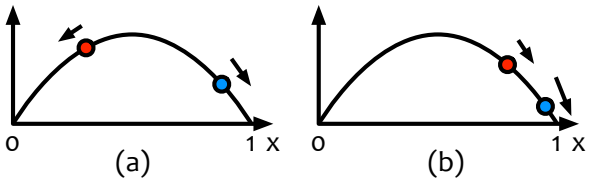
5.2 NetBouncer's latent factor model

We have formulated a latent factor model for the link failure inference problem. Under the constraint $x_i \in [0,1], \forall i$, the objective function to be minimized when estimating the latent link probabilities x_is is the sum of squared errors plus a regularization term as

$$\begin{aligned} &\text{minimize} \quad \sum_j (y_j - \prod_{i:\text{link}_i \in \text{path}_j} x_i)^2 + \lambda \sum_i x_i(1-x_i) \\ &\text{subject to} \quad 0 \le x_i \le 1, \forall i \end{aligned} \tag{2}$$

Specialized regularization. In the model, we have designed a specialized regularization term $\sum_i x_i(1-x_i)$ which incorporates our troubleshooting domain knowledge to reduce false positives described in §5.1.

There are two desired characteristics of this regularization term: *(a)*, it has a two-direction penalty; *(b)*, because of the quadratic term, the closer to 1 the greater the slope.



The characteristic (a) separates the bad links and the good links, as it tends to move the link probability toward 0 or 1. The insight behind this is that the regularization term tends to "pull" the good links to be better, while "push" the bad links to be worse, while the product of link probabilities will stay approximately the same. It helps resolve the false positive cases (e.g., Figure 4 in §5.1) where the imperfect measurement involves a bad link (x_1) and a good link (x_3).

The characteristic (b) mitigates the accidental packet loss and noisy measurements, which helps endorse most links (good) and assign the blame to only a small number of links (bad). The intuition of this characteristic is that (i) most of the links are good, and (ii) the larger the success probability (x_i closer to 1) the more likely the loss is an accidental loss or an inaccurate/noisy measurement. In response, when one x_i is closer to 1, the regularization term provides stronger strength (greater slope) to "pull" this x_i to be 1 (i.e., a good link).

As for the standard penalties, some (e.g., L_1 and L_2) only promote one-direction shrinkage; Other two-direction penalties (e.g., entropy) are inefficient in terms of analytical solution and numerical optimization (our regularization term leads to a neat analytical solution and an associated efficient minimization algorithm).

Non-convex representation. In our model (Equation 2), we use a non-convex representation which, in practice, has better performance than its corresponding convex representation.

From the theoretical perspective, convexity is a desired property that guarantees the convergence to the global optimal solution. The convex representation can be obtained by applying a logarithm transformation to Equation 2 (similar model used by Netscope[18]). It converts the multiplication equations to linear equations and results in a convex problem.

However, our experiments (§6.4) show that the non-convex representation has better performance. The reason is that the convex representation suffers from a scale change and skewed log function (e.g., machine epsilon, no $log(0)$ exists), and thus does not work well numerically in practice.

5.3 Algorithm for link failure inference

Given the above optimization problem, we adopt coordinate descent (CD), an optimization method, to solve the link failure inference problem. This algorithm is depicted in Figure 5 (the pseudocode and complexity analysis are in appendix §B).

Coordinate descent leverages the sparse characteristic of the real-world network which results in an efficient algorithm. By sparsity, we mean that, in a data center network, each link is included by only a few paths comparing to the whole path

Define:

$X \leftarrow all\ x_i, \quad Y \leftarrow all\ y_i$

$f(X,Y) \leftarrow \sum_j (y_j - \prod_{i:\text{link}_i \in \text{path}_j} x_i)^2 + \lambda \sum_i x_i(1-x_i)$

1: **procedure** DETECTBADLINKS(Y)
2: $X \leftarrow$ INITLINKPROBABILITY(Y) // line 12
3: $L_0 \leftarrow f(X,Y)$ // initial value for target function f
4: **for** iteration $k = 1, \cdots,$ MaxLoop :
5: **for** each x_i in X :
6: $x_i \leftarrow \underset{x_i}{argmin}\ f(X,Y)$
7: project x_i to $[0,1]$
8: $L_k \leftarrow f(X,Y)$
9: **If** $L_{k-1} - L_k < \varepsilon$ **then** break the loop
10: **return** $\{(i,x_i) | x_i \leq bad\ link\ threshold\}$
11:
12: **procedure** INITLINKPROBABILITY(Y)
13: $X \leftarrow \{\}$
14: **for** link_i in *links* :
15: // initialize link success probability
16: $x_i \leftarrow avg(\{y_j \mid \text{link}_i \in \text{path}_j\})$
17: **return** X

Figure 5: Coordinate Descent for regularized least squares with constraints. x_i is the success probability of link_i; y_j is the success probability of path_j; ε is the threshold of path error; λ is the tuning parameter of regularization. X is the set of all x_is which changes when x_is are updated.

set. Consequently, for updating the success probability of a link, it is more efficient to leverage the information of the paths relevant to this specific link, which is exactly how CD works.

Why not SGD? Stochastic Gradient Descent (SGD) [52] is a classical and universal optimization algorithm. Nevertheless, it fails to meet our throughput requirement due to the huge amount of data (hundreds GB per hour) generated by the real-world data centers (we've tried to improve SGD to our best, such as lazy-update optimization [8, 32, 38]). By unable to meet our requirement, we mean that SGD cannot produce the failure links within a certain time budget (in practice, 5min for one probing epoch, see §7.2). Our experiments show that CD converges more than one order of magnitude faster than SGD (§6.4).

There are two reasons why SGD is much slower. First, SGD iterates over all the paths, and during each loop for a particular path, all link probabilities are updated simultaneously due to the regularization term. Such updating does not take advantage of the path's sparse representation (only a few links are related to one path) and includes all the links no matter whether they are on the paths or not, and thus is not efficient.

Second, SGD suffers from the boundary constraints. When the link probability is supposed to cross the boundary 0 or 1 according to its current update, we have to clip it and consequently lose information about its gradient.

6 Simulation studies

We run simulations to demonstrate the following: that NetBouncer's probing plan is sufficient (§6.2); that NetBouncer's device failure detection is effective (§6.3); that NetBouncer's design choices are justified (§6.4), and that NetBouncer performs well comparing with other designs (§6.5).

6.1 Simulation setup

We build our simulation on top of a python trace generator whose outputs are path probing data consumed by NetBouncer's processor. The trace generator simulates the path probings on a standard three-layer Clos network [2, 48] including 2.8 thousand switches (48-port switches), 27.6 thousand servers and 82.9 thousand links.

We follow the settings from previous work [18, 41], while change the specific loss rates to fit the data center environment [1]: for faulty links/devices, the packet drop probabilities are between 0.02 and 1; and for non-lossy links, the packet drop probabilities are between 0 and 0.001 (to simulate noise). In addition, we randomly choose 10 devices as faulty devices.

For each probing path, 100 packets are sent. Whether a packet will be successfully transmitted is determined by generating a random number uniformly between 0 and 1. NetBouncer considers a link good if its estimated success probability is greater than 0.999 (because the noise rate is 0.001). All the experiment results are the averages of 10 executions.

6.2 Probing plan

We perform both NetBouncer's probing plan (§4.3) and a *hop-by-hop probing plan* in this experiment. Hop-by-hop probing plan is a probing plan that sends packets from every server to every relevant switch. Because of its exhausted probing, hop-by-hop probing plan is able to identify all the links, but with very high cost.

The results of hop-by-hop probing plan and NetBouncer's probing plan are listed in columns "Hop-by-hop" and "NetBouncer" of Figure 6 respectively. NetBouncer's probing plan achieves the same performance as hop-by-hop probing (there are minor differences with 10% faulty links, which come from the randomness of faulty link selection), while it remarkably reduces the number of paths to be probed.

6.3 Device failure detection

In §4.2, we demonstrate that if all the paths through a specific device are dropping packets, we cannot uniquely infer the link success probabilities (no unique solution exists), which motivates our device failure detection algorithm (§4.4). To

[1] We change the packet drop probability of any faulty link from $[0.05,1]$ to $[0.02,1]$, and that of any non-lossy link from $[0,0.002]$ to $[0,0.001]$, which makes the failure detection more challenging.

Faulty	Hop-by-hop		w/o DFD		Convex		L_1 (λ=0.5)			L_1 (λ=1)			L_1 (λ=2)			NetBouncer (λ=1)		
link%	#FN	#FP	#FN	#FP	#FN	#FP	#FN	#FP	Err	#FN	#FP	Err	#FN	#FP	Err	#FN	#FP	Err
0.1%	0	0	135.3	0	0	46.9	0	48.5	0.01	0	0	0.03	0.3	0	0.14	0	0	0.01
1%	0	0	164.0	0	1.9	522.7	0	81.1	0.07	0	0	0.32	1.1	0	1.41	0	0	0.11
10%	0.6	0	123.3	0	257.6	4.1k	0	695.7	0.91	0.1	0.6	3.80	25.6	0	15.88	0.3	0.2	1.43

Figure 6: Simulation experiment results on variants of NetBouncer with setup in §6.1. "Faulty link%" indicates the proportion of faulty links over all links. "#FN" and "#FP" indicates the number of false negatives and false positives. "Err" is the estimation error (the smaller the better, see the definition in §6.4). As for the results, "Hop-by-hop" represents the experiment using hop-by-hop probing (§6.2); "w/o DFD" represents the results without faulty device detection (§6.3); "Convex" indicates the experiment on convex representation; "L_1" represents the experiments using standard regularization with different parameters (§6.4). And, the final column "NetBouncer" is the performance of NetBouncer.

understand the necessity of device failure detection, we evaluate NetBouncer without its device failure detection (DFD) under different faulty link ratios. Column "w/o DFD" in Figure 6 shows the results.

Without the help of faulty device detection, NetBouncer produces many false negatives. There are two main reasons: First, without faulty device detection, the links associated with the faulty devices have an infinite number of solutions (§4.2). Hence, NetBouncer may end up with an arbitrary solution. Second, the regularization in NetBouncer tends to aggregate the path failures and assign the blames to a small number of links, which is reasonable for link failure detection, but not for faulty device cases, because all the links connected to a faulty device are faulty and should be blamed.

6.4 NetBouncer design choices

To verify NetBouncer's design choices in §5.2 and §5.3, we compare NetBouncer to its variants with alternative choices.

Convex vs. non-convex. We implement the convex version of NetBouncer with similar regularization by applying a logarithmic transformation to Equation 2, and design an algorithm similar[2] to Figure 5 to solve this convex problem.

From the results in column "Convex" of Figure 6, we can see that the convex representation has both false positives and false negatives, which mainly derive from its skewed log scale and boundary clipping. For example, when $x=0$, $log(x)$ is invalid, so we have to assume $x=$1e-8 as an approximation.

L_1 vs. specialized regularization. We now evaluate the effectiveness of NetBouncer's specialized regularization by comparing it with a standard regularization L_1. The L_1 regularization term is defined as $-\sum_i x_i$ [18]. In order to compare the estimation accuracy, we use the squared estimation error of link probabilities as a metric, which is defined as $\sum_i (x_i - x'_i)^2$ (x_i is the success probability from "ground truth" and x'_i is its estimate). The smaller the error metric ("Err" in Figure 6), the more accurate the estimation.

The results using the L_1 regularization with different λs are presented in Figure 6. From the experiments, we find that the specialized regularization obtains similar numbers of false discoveries to L_1 under the best tuning parameter λ in terms of failure diagnosis, while it achieves much lower estimation error when fitting link probabilities.

[2]There are slight differences for the boundary handling due to the log scale of the convex model.

OptMethod	Learning rate	#round	Time(s)
CD	–	4	14.8
SGD-lazy	0.001	145	513.3
SGD-lazy	0.005	45	157.5
SGD-lazy	0.01	161	569.9

Figure 7: Comparison of CD and SGD. The faulty link% of the workload is 0.1% and λ is 1. To help SGD converge, we have relaxed the convergence condition for SGD ($\varepsilon = 0.0001$ for CD, $\varepsilon = 0.001$ for SGD). "#round" is the number of rounds to converge. "Time(s)" is the total time of execution in seconds.

SGD vs. CD. Although SGD is broadly used as an off-the-shelf method, we adopt CD, a more efficient method for NetBouncer based on the sparse structure of our problem. To make a fair comparison, we implement an SGD with the lazy-update optimization [8, 32, 38], which is more efficient than a naive implementation.

Figure 7 summarizes the performance of CD and SGD. It shows that CD converges much faster than SGD. The time spent on each round of CD and SGD are similar, but CD converges in significantly fewer rounds, which validates our analysis.

Regularization parameter λ. The results of NetBouncer are affected by the tuning parameter λ in the regularization affects. Intuitively, λ balances the fitting error and false discoveries. A large λ may trigger false negatives, while a small λ may leave false positives.

To illustrate how the results change as λ varies, we run NetBouncer with different λ values on the network with 1% faulty links. The numbers of false discoveries are shown in Figure 8 where the x-axis is in log-scale. The results demonstrate the trade-off between false positives and false negatives from choosing λ.

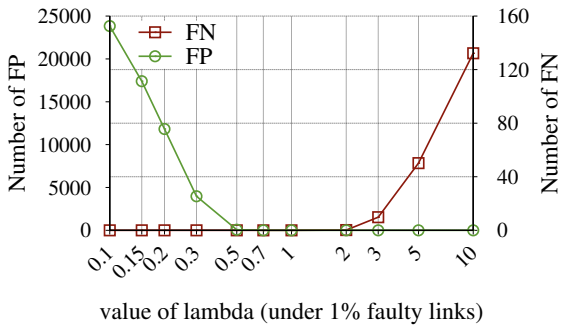


Figure 8: Number of false positives and false negatives for different λ.

Faulty link%	0.1%		1%		10%	
	#FN	#FP	#FN	#FP	#FN	#FP
NetBouncer	0	0	0	0	0.3	0.2
deTector (0.6)	187.5	6.0k	215.5	7.2k	204.0	22.8k
deTector (0.9)	204.5	0.7	191.5	0.4	208.0	21.7
NetScope (0.1)	0	9.1k	3.0	10.8k	167.5	12.6k
NetScope (1)	0.3	43.7	10.2	395.5	319.5	3.8k
NetScope (10)	28.7	6.3	291.5	86.7	2.4k	1.2k
KDD14	7.8	21.0	76.6	433.2	213.8	3.0k

Figure 9: Compare NetBouncer with existing schemes. The number in parentheses of "deTector" is a tuning parameter to filter false positives (*hit ratio* in paper[44]). The number in parentheses of "NetScope" is its regularization parameter (ω in paper[18]).

6.5 Comparison with existing systems

In this section, we compare NetBouncer with three existing systems: deTector [44], NetScope [18], and KDD14 [25]. Note that we only compare the failure inference part on the processor (step ③ in Figure 1), that is inferring failures from the path measurement data. To the best of our knowledge, deTector is the state-of-the-art heuristic algorithm after a long line of work [16, 31, 43]. NetScope improves the previous work [41, 49] and takes data inconsistency into account. KDD14 applies statistical techniques to data center failure localization. They are the most relevant network diagnosis algorithms, and thus the most appropriate benchmarks for NetBouncer.

The original NetScope is designed for troubleshooting on Internet which is an underdetermined system, whereas data center network is an overdetermined system. As a result, we extend the NetScope algorithm following its core idea – "L_1-norm minimization with non-negativity constraints" (§III.D in [18]), and apply it to the logarithmic transformation of Equation 1. For KDD14, we assume that the routing path of each packet is known, which is an improvement over the original version.

Using the same setup in §6.1, we run experiments on NetBouncer as well as deTector, NetScope and KDD14 with various faulty link ratios, and present the results in Figure 9.

As a greedy algorithm, deTector is designed to blame the smallest number of links, so that it incurs false negatives when there are faulty devices. On the other hand, deTector also incurs false positives due to noise in the data. Lastly, it uses a tuning parameter (i.e., hit ratio in [44]) to filter false positives, which may result in false negatives as well. This is a trade-off requiring domain knowledge and experiences from the network operators.

Similar to the implementation of NetBouncer's convex representation, NetScope encounters numerical problems mainly due to its logarithmic representation. KDD14 produces both false positives and false negatives resulting from its assumption that there is at most one faulty link among all the paths between two servers, which is unrealistic for a large-scale data center network.

7 Implementation and evaluation

7.1 Implementation

Controller. The NetBouncer Controller is a central place to decide how the agents probe the whole network. It takes the network topology as input and generates a probing plan for servers. The controller has multiple replicas for fault tolerance.

Agent. The NetBouncer Agent runs on servers. It fetches its probing plan from the Controller, which contains the paths to be probed. For each path, the probing plan contains the related parameters including the number of packets to send, the packet size, the UDP source destination port range, the probe frequency, the TTL and ToS values, etc. For each probed path, the Agent generates a record which contains the path, the packet length, the total number of packets sent, the number of packet drops, the RTTs at different percentiles. The CPU and traffic overhead of the agents are both negligible in practice.

Processor. The NetBouncer Processor has two parts. A front-end which is responsible for collecting the records from the NetBouncer Agents, and a back-end Data processor which runs the algorithm. The front-end and back-end run on the same physical server. For load-balance and geo fault tolerance considerations, we run multiple NetBouncer Processors. Each Processor is responsible for a set of data center regions. Within a geolocation, we run four NetBouncer Processor instances behind a software load-balancer VIP. One instance acts as the master and runs the NetBouncer algorithm, the other three are slaves. The VIP is configured in a way so that only the master receives records from the agents.

Result verification and visualization. After localizing failures in the network, NetBouncer provides a result verification tool which the operators can use to issue probing packets on-demand. These on-demand verification packets are sent from the same Agents as the NetBouncer service.

This tool also shows the packet drop history of links for visualization. One failure example is illustrated in Figure 10. It reveals the packet drop history of a link detected by NetBouncer. Users can click the "Quick probe" button to

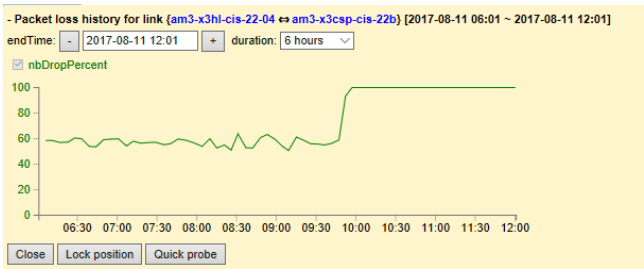


Figure 10: Result verification and visualization tool.

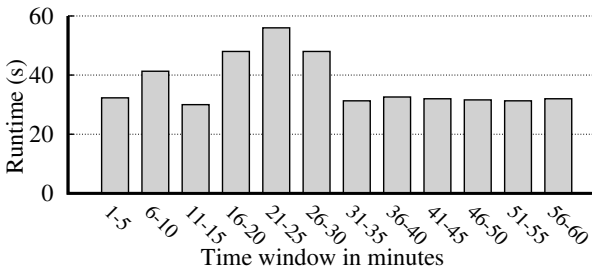


Figure 11: NetBouncer running time on real-world data.

launch probes on-demand for verification. As the figure shows, the packet drop rate of the link changed from 60% to 100% at around 10:00. This is because the NetBouncer report triggered our network repairing service, which in turn shutdown this link to mitigate the incident. The link will be manually inspected, cleaned up, and restored later.

With the help of this verification tool, we gain more confidence in the performance of NetBouncer. For the cases we have verified, we did not experience any false positives. However, NetBouncer is not false negative-free, as discussed in § 9.

Probing epoch. Probing epoch, the interval that NetBouncer does one run of failure inference algorithm, is a critical parameter of the system design. One deceptive intuition is that a shorter probing epoch always leads to faster failure detection. However, in reality, too frequent failure inferences may result in either less confident results or bursts of probing traffic.

From our experiences, determining the probing epoch is a three-way trade-off among inference accuracy, probing resource cost, and failure localization speed. On one hand, with fixed probing frequency, inferring failures too often ends up with less probing data for each inference, and hence may harm the accuracy. On the other hand, by increasing the probing frequency on servers, the accuracy is able to keep unchanged. Nevertheless, it may cause probing traffic bursts, which may introduce network congestion, ramping up the risk of instability of the whole network.

Considering the previous tradeoffs, NetBouncer chooses 5 minutes as one probing epoch. Yet, it is plausible to sacrifice the other two factors to significantly shorten the probing time.

7.2 Data processor runtime evaluation

In this section, we evaluate the performance of Data Processor, which is executed on a machine having Intel Xeon E5 2.4GHz CPU with 24 cores (48 logical cores) and 128GB memory. The operating system is Windows Server 2016.

We have around 30 regions in total. The Data Processor uses one thread to process the data for one region. Data processing in the Data Processor is therefore naturally parallelized.

We run NetBouncer on one-hour real-world data trace on November 5, 2016, which is 130GB in size and covers tens of global data centers (other hours have similar results). In production, NetBouncer detects the faulty links in the hopping window of every 5 minutes. We follow the same detection frequency in this experiment.

As we show in Figure 11, the min, average, and max running times are 30.0s, 37.3s, and 56.0s, respectively. The max processing time happened at window 21-25, in which our detection algorithm converged after 9 iterations, whereas most of the rest windows finished in 4 iterations. We did the investigation and found that the additional iterations are caused by a few "hard-to-solve" faulty links, when the faulty links appear in the same local sub-graph. In that case, our algorithm needs to go back and forth with several more iterations to converge. In all the cases, the time-to-detection (TTD) for failures is within 60 seconds.

We also study the time spent on each of the stages. On average, NetBouncer's algorithm with regularization (§5.2) costs 54.9% of the CPU time; other stages – faulty device detection (§4.4), data cleaning and data structure setup – take 12.4%, 23.8% and 8.9%, respectively. NetBouncer consumes about 15-20 GB memory during the processing which is only a small portion of the 128G memory of the server.

Overall, a single NetBouncer Processor instance can handle multiple regions with tens of thousands of switches and more than one million network links, with spare capacity for future growth.

8 Deployment experiences

NetBouncer has been running in Microsoft Azure for three years. In this section, we're going to share our deployment experiences and some representative failures NetBouncer detected.

NetBouncer deployment, before vs. after. Before NetBouncer was deployed, gray failures could last hours to days. Since no clue was provided, operators had to pinpoint the failure via trial and error in a large region of the network based on their troubleshooting experiences.

After NetBouncer went online, it has successfully reduced the detection time from hours to minutes, and further shortening is also possible (see the probing epoch in §7.1). Moreover, NetBouncer greatly deepened our understanding of the reasons why packet drops happen, including silent packet drops, packet blackholes, link congestion, link flapping, BGP routing flapping, switch unplanned reboot, etc. For example, it

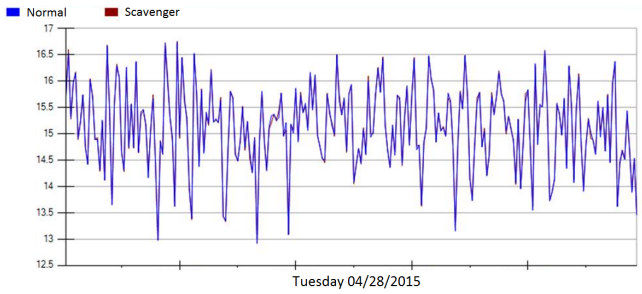


Figure 12: The packet drop probability detected by NetBouncer. This is a silent packet drop case.

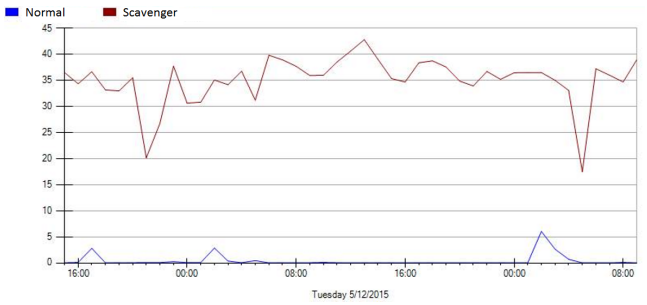


Figure 13: The packet drop probability detected by NetBouncer, caused by link polarization.

once caught a case where a switch was periodically changing its state between normal and 100% packet drops. It turned out that the switch was rebooted continuously by a buggy firmware upgrade procedure.

Next, we present three representative cases in production detected by NetBouncer, which otherwise would be extremely difficult or tricky to locate.

Case 1: spine router gray failure. The first case is one of the most challenging scenarios that motivated the development of NetBouncer: gray failures, where switches silently drop packets without any signals. We had an incident where a spine switch was dropping packets silently, because of an issue in one of this switch's linecard hardware. Many of our customers experienced packet drops and latency increases. The wide impact was also detected by our end-to-end latency service Pingmesh [23]. It was clear that one or more spine switches were dropping packets. But we could not tell which one.

We obtained the link loss rate history of all the spine links using NetBouncer. Figure 12 shows the packet drop history of the lossy link. We found that this link was constantly dropping packets with around 15% packet drop probability. It dropped packets without differentiation, as both the "Normal" and "Scavenger" traffic encountered the same dropping rate (the two curves in Figure 12 overlap with each other). In our network, Normal traffic is given priority over Scavenger traffic.

Case 2: polarized traffic. In the second case, NetBouncer caught a network congestion scenario and also helped identify the root-cause of the incident: a switch firmware bug, which polarized the traffic load onto a single link. Figure 13 shows the packet drop probability measured by NetBouncer. Among a huge number of packet drops, we observed that the packets on Scavenger traffic was dropped at a probability around 35%, but the Normal traffic was not affected. Since NetBouncer directly told us the congested link, detection became trivial. We then mitigated this issue by rekeying the ECMP hash function and solved the problem.

Case 3: miscounting TTL. Time to live (TTL) in an IP packet is supposed to be decremented by one through each switch. However, NetBouncer has discovered that when an IP packet passes though a certain set of switches, its TTL is decremented by two. This issue manifests as a "false positive" by misclassifying affected good links as bad links, which is in fact caused by an internal switch fireware bug. Though this miscounting TTL bug hasn't caused harmful consequences on the actual data traffic yet. Nevertheless, such a hidden issue will raise severe latent risk for service reliability and cause huge confusion for troubleshooting.

False negatives and false positives. In practice, we ran into several false negative cases. In one case, we once ran into a DHCP booting failure, in which some servers could send out the DHCP DISCOVER packets, but could not receive the responding DHCP OFFER packets from the DHCP server. NetBouncer did not detect such DHCP packet drops. Resorting to packet capturing, we could identify that the switches did not drop the DHCP OFFER packets, and this problem was caused by the NIC.

In another case, we encountered an incident due to a misconfigured switch ACL, which resulted in packet drops for a very limited set of IP addresses. Since NetBouncer scanned a wide range of IP addresses, so the signal detected by NetBouncer was weak. Similarly, NetBouncer cannot help when some firewall rules were (wrongly) applied to certain applications.

NetBouncer in theory is not false positive-free. However, NetBouncer did not produce false positives in production so far, because of our specialized regularization (§5.2) and a strict criteria (1% packet drops) for reducing false alarms.

9 Discussions

NetBouncer's limitations. NetBouncer has two major limitations. First, NetBouncer makes an assumption that the probing packets experience the same failures as real applications, which may not hold in all cases (as we shown in §8). Our future work is to systematically investigate the false negative conditions and improve the coverage of our probing packets.

Second, theoretically, NetBouncer cannot guarantee zero false positives or zero false negatives. Nevertheless, this is ubiquitous to all real-world monitoring systems, since the measurement data cannot be perfectly accurate. In practice, NetBouncer has produced no false positives and only a few false negatives (both confirmed by the network operators) so far.

Theory vs. practice. Theoretically, NetBouncer's probing plan is proved to be link-identifiable (§4.3). However, in practice, such nice theory property does not guarantee the results to be false positive free or false negative free, which drove us to seek help from machine learning approaches (latent factor model, regularization and CD).

Yet, we argue that the theory result, though not sufficient, is necessary for solving our problem in reality. Without the guidance of the *sufficient probing theorem*, a chosen probing plan might not be link-identifiable, thus the outputs of the machine learning approach can be arbitrary.

Does the independent assumption hold? In NetBouncer's model (§4.1), we assume that the failures are independent. Our experiences reveal that this assumption holds for a large number of scenarios including random packet drops due to cable/fiber bit errors, infrequently hardware gray failures e.g., bit flips in hardware memory, and packet drops caused by software Heisenbugs. Those scenarios share the same characteristic that they are hard to detect and localize. Once detected, they are typically easy to mitigate (by shutting down the problematic links or rebooting the faulty devices).

How does NetBouncer handle congestion packet loss? As stated in overview (§2), NetBouncer targets *non-transient* failures. NetBouncer treats persistent congestion as failure because persistent congestion affects users. NetBouncer filters out transient congestion since it uses minute-level failure detection interval.

10 Related work

Network tomography. Compared with original network tomography approaches [5, 6, 9, 13, 14, 17, 18, 51] which target the Internet-like networks, NetBouncer has different challenges. First, the topology of a data center network is known, but it requires to design a link-identifiable probing plan. Second, the standard of a well-behaving link in the Internet (failure probability $< 2\%$ in [18]) is way lower than that in a data center network (usually, failure probability $< 0.1\%$).

As for the link failure inference algorithm, Tomo [16] and deTector [44] use heuristic algorithms for failure inference. However, these approaches may generate false results due to their heuristic nature. NetScope [18] (as well as NetQuest [49] and LIA [41]) takes data inconsistency into account. NetBouncer also uses a similar approach (i.e., regularization), but in addition we encode our troubleshooting domain knowledge into the model which results in better performance.

What differentiates NetBouncer from other tomography systems is that NetBouncer provides a complete framework targeting data center networks, including probing plan design (§4.3), device failure detection (§4.4) and link failure inference (§5.2) against real-world data inconsistency.

Other failure localization approaches. SNMP and OpenFlow are widely used in nowadays data centers. However, recent research [23, 28] shows that these tools cannot detect gray failures, a type of failure that causes availability breakdowns and performance anomalies in cloud environment.

Herodotou et al. [25] use statistical data mining techniques for data center failure localization. They propose a probabilistic path model since it does not control the path of a probing packet. Whereas, NetBouncer controls the routing paths of the probing packets. Furthermore, they assume there is only one link failure in a path, which leads to false positives and false negatives as we have shown in §6.5.

Sherlock [4] assumes that only few failures exist in the system. It then enumerates all the possible combinations to find the best match. The running time grows exponentially along with number of failures, which is unacceptable for a large-scale network.

Pingmesh [23], NetSonar [55] and NetNORAD [1] use a TCP or UDP agent for end-to-end reachability and traceroute variants (Tcptracert or fbtracert) for path probes. However, the ICMP packets generated from the probes need to be handled by the switch CPUs, hence need to be carefully managed. In addition, NetSonar uses Sherlock [4] algorithm for failure detection, which cannot support many simultaneous failures.

Passive probing [46] uses core switches to tag IDs in the DSCP or TTL field of IP header for path pinpointing. However, DSCP and TTL fields, which are commonly used for QoS, might not be available.

NetPoirot [3] and [10] leverage decision trees for failure diagnosis. The data sources are from end-host application and TCP logs. These approaches can tell whether the problem is from the network or not, but they do not work for our scenario as they do not differentiate ECMP paths.

Network troubleshooting. Several systems [24, 50, 56] have been proposed for network troubleshooting and debugging. These systems typically need to capture packets or collect packet summaries, which are complementary to NetBouncer.

Other troubleshooting systems need either non-trivial modification on software stack [33, 34, 35, 40], or hardware support [37], which cannot be transparently applied to current data centers in production.

11 Conclusion

In this paper, we propose the design and implementation of NetBouncer, an active probing system which infers the device and link failures from the path probing data. We demonstrate that NetBouncer's probing plan design, device failure detection, and link failure inference perform well in practice. NetBouncer has been running in Microsoft Azure's data centers for three years, and has helped mitigate numerous network incidents.

References

[1] ADAMS, A., LAPUKHOV, P., AND ZENG, J. H. Netnorad: Troubleshooting networks via end-to-end probing. https://code.facebook.com/posts/1534350660228025/netnorad-troubleshooting-networks-via-end-to-end-probing/, Febrary 2016.

[2] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. In *ACM SIGCOMM* (2008).

[3] ARZANI, B., CIRACI, S., LOO, B. T., SCHUSTER, A., AND OUTHRED, G. Taking the blame game out of data centers operations with netpoirot. In *ACM SIGCOMM* (2016).

[4] BAHL, P., CHANDRA, R., GREENBERG, A., KANDULA, S., MALTZ, D. A., AND ZHANG, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM* (2007).

[5] BATSAKIS, A., MALIK, T., AND TERZIS, A. Practical passive lossy link inference. In *Passive and Active Measurement Workshop* (2005).

[6] CÁCERES, R., DUFIELD, N. G., HOROWITZ, J., AND TOWSLEY, D. Multicast-based inference of network-internal loss characteristics. *IEEE Trans. Inform. Theory 45* (November 1999).

[7] CAO, J., XIA, R., YANG, P., GUO, C., LU, G., YUAN, L., ZHENG, Y., WU, H., XIONG, Y., AND MALTZ, D. Per-packet load-balanced, low-latency routing for clos-based data center networks. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2013).

[8] CARPENTER, B. Lazy sparse stochastic gradient descent for regularized multinomial logistic regression. *Alias-i, Inc., Tech. Rep* (2008), 1–20.

[9] CASTRO, R., COATES, M., LIANG, G., NOWAK, R., AND YU, B. Network tomography: Recent developments. *Statistical Science 19* (Auguest 2004).

[10] CHEN, M., ZHENG, A. X., LLOYD, J., JORDAN, M. I., AND BREWER, E. Failure diagnosis using decision trees. In *Proceedings of the First International Conference on Autonomic Computing* (2004).

[11] CHEN, Y., BINDEL, D., AND KATZ, R. H. Tomography-based overlay network monitoring. In *ACM IMC* (2003).

[12] CHUA, D. B., KOLACZYK, E. D., AND CROVELLA, M. Efficient monitoring of end-to-end network properties. In *IEEE International Conference on Computer Communications (INFOCOM)* (2005).

[13] COATES, M., AND NOWAK, R. Network Loss Inference Using Unicast End-to-End Measurement. In *Proc.ITC Conf IP Traffic, Modeling and Management* (2000).

[14] CUNHA, I., TEIXEIRA, R., FEAMSTER, N., AND DIOT, C. Measurement methods for fast and accurate blackhole identification with binary tomography. In *ACM IMC* (2009).

[15] DE GHEIN, L. *MPLS fundamentals*. Cisco Press, 2016.

[16] DHAMDHERE, A., TEIXEIRA, R., DOVROLIS, C., AND DIOT, C. Netdiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2007).

[17] DUFFIELD, N. Network tomography of binary network performance characteristics. *IEEE Transactions on Information Theory 52* (Dec 2006).

[18] GHITA, D., NGUYEN, H., KURANT, M., ARGYRAKI, K., AND THIRAN, P. Netscope: Practical network loss tomography. In *IEEE International Conference on Computer Communications (INFOCOM)* (2010).

[19] GILL, P., JAIN, N., AND NAGAPPAN, N. Understanding network failures in data centers: Measurement, analysis, and implications. In *ACM SIGCOMM* (2011).

[20] GOVINDAN, R., MINEI, I., KALLAHALL, M., KOLEY, B., AND VAHDAT, A. Evolve or die: High-availability design principles drawn from google's network infrastructure. In *ACM SIGCOMM* (2016).

[21] GUILBAUD, N., AND CARTLIDGE, R. Localizing packet loss in a large complex network (ppt). https://www.nanog.org/meetings/nanog57/presentations/Tuesday/tues.general.GuilbaudCartlidge.Topology.7.pdf.

[22] GUNAWI, H. S., HAO, M., SUMINTO, R. O., LAKSONO, A., SATRIA, A. D., ADITYATAMA, J., AND ELIAZAR, K. J. Why does the cloud stop computing? lessons from hundreds of service outages. In *SoCC* (2016).

[23] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., ET AL. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM* (2015).

[24] HANDIGOL, N., HELLER, B., JEYAKUMAR, V., MAZIÈRES, D., AND MCKEOWN, N. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2014).

[25] HERODOTOU, H., DING, B., BALAKRISHNAN, S., OUTHRED, G., AND FITTER, P. Scalable near real-time failure localization of data center networks. In *KDD* (2014).

[26] HU, S., CHEN, K., WU, H., BAI, W., LAN, C., WANG, H., ZHAO, H., AND GUO, C. Explicit path control in commodity data centers: Design and applications. *IEEE/ACM Transactions on Networking* (2016).

[27] HUANG, P., GUO, C., LORCH, J. R., ZHOU, L., AND DANG, Y. Capturing and enhancing in situ system observability for failure detection. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2018).

[28] HUANG, P., GUO, C., ZHOU, L., LORCH, J. R., DANG, Y., CHINTALAPATI, M., AND YAO, R. Gray failure: The Achilles' heel of cloud-scale systems. In *Workshop on Hot Topics in Operating Systems (HotOS)* (2017).

[29] JYOTHI, S. A., DONG, M., AND GODFREY, P. Towards a flexible data center fabric with source routing. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (2015).

[30] KOHAVI, R., AND LONGBOTHAM, R. Online experiments: Lessons learned. *IEEE Computer* (September 2007).

[31] KOMPELLA, R. R., YATES, J., GREENBERG, A., AND SNOEREN, A. C. Ip fault localization via risk modeling. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2005).

[32] LANGFORD, J., LI, L., AND ZHANG, T. Sparse online learning via truncated gradient. *Journal of Machine Learning Research 10*, Mar (2009), 777–801.

[33] LENERS, J. B., GUPTA, T., AGUILERA, M. K., AND WALFISH, M. Improving availability in distributed systems with failure informers. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2013).

[34] LENERS, J. B., GUPTA, T., AGUILERA, M. K., AND WALFISH, M. Taming uncertainty in distributed systems with help from the network. In *European Conference on Computer Systems (EuroSys)* (2015).

[35] LENERS, J. B., WU, H., HUNG, W.-L., AGUILERA, M. K., AND WALFISH, M. Detecting failures in distributed systems with the falcon spy network. In *ACM Symposium on Operating Systems Principles (SOSP)* (2011).

[36] LI, H., GAO, Y., DONG, W., AND CHEN, C. Taming both predictable and unpredictable link failures for network tomography. In *Proceedings of the ACM Turing 50th Celebration Conference-China* (2017).

[37] LI, Y., MIAO, R., KIM, C., AND YU, M. Lossradar: Fast detection of lost packets in data center networks. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2016).

[38] LIPTON, Z. C., AND ELKAN, C. Efficient elastic net regularization for sparse linear models. *arXiv preprint arXiv:1505.06449* (2015).

[39] MA, L., HE, T., LEUNG, K. K., SWAMI, A., AND TOWSLEY, D. Identifiability of link metrics based on end-to-end path measurements. In *Proceedings of the 2013 conference on Internet measurement conference* (2013).

[40] MOSHREF, M., YU, M., GOVINDAN, R., AND VAHDAT, A. Trumpet: Timely and precise triggers in data centers. In *ACM SIGCOMM* (2016).

[41] NGUYEN, H. X., AND THIRAN, P. Network loss inference with second order statistics of end-to-end flows. In *ACM SIGCOMM* (2007).

[42] PADMANBHAN, V., QIU, L., AND WANG, H. Server-based inference of internet performance. In *In Proc. of IEEE INFOCOM* (2003).

[43] PATI, Y. C., REZAIIFAR, R., AND KRISHNAPRASAD, P. S. Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In *Signals, Systems and Computers, 1993. 1993 Conference Record of The Twenty-Seventh Asilomar Conference on* (1993).

[44] PENG, Y., YANG, J., WU, C., GUO, C., HU, C., AND LI, Z. detector: a topology-aware monitoring system for data center networks. In *USENIX Annual Technical Conference* (2017).

[45] POTHARAJU, R., AND JAIN, N. When the network crumbles: An empirical study of cloud network failures and their impact on services. In *SoCC* (2013).

[46] ROY, A., ZENG, H., BAGGA, J., AND SNOEREN, A. C. Passive realtime datacenter fault detection and localization. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

[47] SIMPSON, W. IP in IP Tunneling, 1995. RFC 1853.

[48] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., ET AL. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *ACM SIGCOMM* (2015).

[49] SONG, H. H., QIU, L., AND ZHANG, Y. Netquest: a flexible framework for large-scale network measurement. In *ACM SIGMETRICS Performance Evaluation Review* (2006).

[50] TAMMANA, P., AGARWAL, R., AND LEE, M. Simplifying datacenter network debugging with pathdump. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[51] VARDI, Y. Netowrk tomography: Estimating source-destination traffic intensities from link data. *Journal of the American Statistical Association 91* (March 1996).

[52] WIKIPEDIA. Stochastic gradient descent. `https://en.wikipedia.org/wiki/Stochastic_gradient_descent`.

[53] WRIGHT, S. J. Coordinate descent algorithms. *Mathematical Programming 151*, 1 (2015), 3–34.

[54] ZENG, H., KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Automatic test packet generation. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2012).

[55] ZENG, H., MAHAJAN, R., MCKEOWN, N., VARGHESE, G., YUAN, L., AND ZHANG, M. Measuring and troubleshooting large operational multipath networks with gray box testing. Tech. Rep. MSR-TR-2015-55, Microsoft Research, 2015.

[56] ZHU, Y., AND ET AL. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM* (2015).

A Proof of sufficient probing theorem

Proof. **Sufficient condition**: For any link with its success probability x_0, we consider the only two possibilities:

(1) If a path with success probability 1 includes this link, then $x_0 = 1$.

(2) Otherwise, the condition guarantees that at least one link with success probability 1 connects the upper (lower) node of this link to a node in the upper (lower) layer. According to the way we probe, at least one path includes this link so that all the other links in this path have success probability 1. Thus, we will find an equation $x_0 = y_0$ in the path success probability equations.

Necessary condition: Assume the condition does not hold, then there exists at least one node so that all of its links have success probability not 1. Therefore, there exists a subgraph including this node connected with n nodes in the upper layer and m nodes in the lower layer, which is separable from the rest of the whole graph.

Given that the success probabilities of all the other links in the rest of the whole graph are known, we try to solve this subgraph and consider the only three possibilities:

(1) If all subpath probabilities are not 0 in this subgraph, then we can transform the path success probability equations of this subgraph to the linear equations

$$\log y_{i,n+j} = \log x_i + \log x_{n+j}, 1 \le i \le n; 1 \le j \le m$$

where the rank of corresponding matrix is $n + m - 1$, less than the number of linear equations. As a result, no unique solution is available.

(2) If some but not all subpath probabilities are 0 in this subgraph, we only focus on the nonzero equations, which do not have a unique solution due to the redundancy either.

(3) If all subpath probabilities are 0 in this subgraph, we simply cannot distinguish the solution where all links are 0 or just some are 0.

To sum up, the solution is not unique in the subgraph even when the rest of the whole graph is solved. Therefore, it is impossible to get a unique solution for the whole graph. □

B Failure inference algorithm and complexity analysis

Algorithm 1 describes the full version of NetBouncer's failure inference algorithm, Coordinate Descent for regularized least squares with constraints.

We analyze the complexity of Algorithm 1 as follows. Assume that each link is included by C paths on average. The time complexity is $O(CM)$ for link initialization. In each iteration, the time complexity is $O(CM)$ for link updating, and $O(N+M)$ for convergence checking. Suppose there are K iterations until convergence, the total time complexity is $O(CM + KCM + KN + KM) = O(KCM + KN)$. The space complexity is $O(N)$ for all path rates, and $O(M)$ for all link rates, and $O(N)$ for the mapping from the paths to the links they include, and $O(CM)$ for the mapping from the links to the paths including them. Then the total space complexity is $O(N+M+N+CM) = O(N+CM)$.

Algorithm 1 Coordinate Descent for regularized least squares with constraints.

Require: N = number of paths, M = number of links, y_j = sample success probability of path j, n_j = sample size of path j, K = maximal number of iterations, ε = threshold of path error, λ = tuning parameter of regularization.

initialize $x_i^{(0)} = \sum_{j:x_i \in y_j} (n_j y_j) / \sum_{j:x_i \in y_j} n_j$, $i \in [1,M]$
for iteration $k = 1,\cdots,K$ **do**
  $x_i^{(k)} = x_i^{(k-1)}$, $i \in [1,M]$
  for link $i = 1,\cdots,M$ **do**
    $R_i^{(k)} = 2 \sum_{j:x_i \in y_j} (\prod_{\ell:\ell \neq i, x_\ell \in y_j} x_\ell^{(k)})^2 - 2\lambda$
    $S_i^{(k)} = 2 \sum_{j:x_i \in y_j} (y_j \prod_{\ell:\ell \neq i, x_\ell \in y_j} x_\ell^{(k)}) - \lambda$
    $T_i^{(k)} = S_i^{(k)} / R_i^{(k)}$
    if $R_i^{(k)} = 0$ **then**
      if $S_i^{(k)} > 0$ **then** $x_i^{(k)} = 1$
      else $x_i^{(k)} = 0$
    else if $R_i^{(k)} > 0$ **then**
      if $T_i^{(k)} > 1$ **then** $x_i^{(k)} = 1$
      else if $T_i^{(k)} < 0$ **then** $x_i^{(k)} = 0$
      else $x_i^{(k)} = T_i^{(k)}$
    else
      if $T_i^{(k)} > 1/2$ **then** $x_i^{(k)} = 0$
      else $x_i^{(k)} = 1$
  $L^{(k)} = \sum_j (y_j - \prod_{i:x_i \in y_j} x_i^{(k)})^2 + \lambda \sum_i x_i^{(k)} (1 - x_i^{(k)})$
  If $L^{(k-1)} - L^{(k)} < \varepsilon$ **then** break the loop
return $\{x_i^{(k)} | i \in [1,M]\}$

C Acknowledgement

We thank Jain Shalabh Jain and Pradeepkumar Mani for their contributions to the early NetBouncer system.