# *deTector*: a Topology-aware Monitoring System for Data Center Networks

Yanghua Peng
*The University of Hong Kong*

Ji Yang
*Xi'an Jiaotong University*

Chuan Wu
*The University of Hong Kong*

Chuanxiong Guo
*Microsoft Research*

Chengchen Hu
*Xi'an Jiaotong University*

Zongpeng Li
*University of Calgary*

## Abstract

Troubleshooting network performance issues is a challenging task especially in large-scale data center networks. This paper presents *deTector*, a network monitoring system that is able to detect and localize network failures (manifested mainly by packet losses) accurately in near real time while minimizing the monitoring overhead. *deTector* achieves this goal by closely coupling fault detection and localization, as well as carefully selecting probe paths, so that packet losses can be localized only using end-to-end observations without the help of additional tools (*e.g.*, tracert). In particular, we quantify the desirable properties of probe path matrix, *i.e.*, coverage and identifiability, and leverage an efficient greedy algorithm with a good approximation ratio and fast speed to select probe paths. We also propose a loss localization method according to loss patterns in a data center network. Experimental evaluation on a Fattree testbed and supplementary large-scale simulation validate the scalability, feasibility and effectiveness of *deTector*.

## 1 Introduction

A variety of services are hosted in large-scale data centers today, *e.g.*, search engines, video streaming, social networks and file sharing. To support these services with high quality, data center networks (DCNs) are carefully designed to efficiently connect thousands of network devices together, *e.g.*, a 64-ary Fattree [4] DCN has more than 60,000 servers and 5,000 switches. However, due to the large network scale, frequent upgrades and management complexity, failures in DCNs are the norm rather than the exception [16], such as routing misconfigurations, link flaps, etc. Among these failures, those leading to user-perceived performance issues (*e.g.*, packet losses, latency spikes) are among the first priority to be detected and eliminated promptly [22, 21, 16], in order to maintain high quality of service (QoS) for users (*e.g.*, no more than a few minutes of downtime per month [16]) and to increase revenue for operators.

Rapid failure recovery is not possible without a good network monitoring system. There have been a number of systems proposed in the past few years [33, 21, 34, 44]. Several limitations still exist in these systems that prohibit fast failure detection and localization.

*First*, existing monitoring systems are not generic enough to detect various types of losses. Traditional passive monitoring approaches, such as querying the device counter via SNMP or retrieving information via device CLI when users have perceived some issues, can detect clean failures such as link down, line card malfunctions. However, gray failures may occur, *i.e.*, faults not detected or ignored by the device, or malfunctioning not properly reported by the device due to some bugs [34]. Active monitoring systems (*e.g.*, Pingmesh [21], NetNORAD [34]) can detect such failures by sending end-to-end probes, but they may fail to capture failures that cause low rate losses, due to ECMP in data center (§2).

*Second*, probe systems such as Pingmesh and NetNORAD inject probes between each pair of servers without selection, which may introduce too much bandwidth overhead. In addition, they typically treat the whole DCN as a black box, and hence require many probes to cover all parallel paths between any server pair with high probability.

*Third*, failures in the network can be reported in these active monitoring systems, but the exact failure locations cannot be pinpointed automatically. The network operator typically learns a suspected source-destination server pair once packet loss happens. Then he needs to resort to additional tools such as tracert to verify the issue and locate the faulty spot. However, it may be difficult to play back the issues due to transient failures. Hence this diagnosis approach (*i.e.*, separation of detection and localization) may take several hours or even days to pinpoint the fault spot [16], yet ideally the failures should be repaired as fast as possible before users complain.

A desirable monitoring system in a DCN should meet three criteria: exhaustive failure detection (*i.e.*, detecting as many types of losses as possible), low overhead and real-time failure localization. In this paper, we seek to investigate the following question: if we are aware of the network topology of a DCN, can we design a much better network monitoring system that achieves all these goals? Our answer is *deTector*, a topology-aware network monitoring system that we design, implement and evaluate following the three design objectives. The secret weapon of *deTector* is a carefully designed probe matrix (§4), which achieves good link coverage, identifiability and evenness. *deTector* is designed to detect and localize network failures manifested by user-perceptible performance problems such as packet losses and latency spikes in large-scale data centers. We mainly focus on packet loss in this paper, but *deTector* can also handle latency issues by treating a round trip time (RTT) larger than a threshold as a packet loss. Throughout the paper, we use "failure localization", "fault localization" and "loss localization" interchangeably. Specifically, we make the following contributions in developing *deTector*.

▷ As compared to the existing active monitoring systems adopting end-to-end probes (*e.g.*, Pingmesh [21], NetNORAD [34]), we treat each switch instead of the whole network as a blackbox, *i.e.*, our system is topology-aware and we use source routing to control the probing path. In order to achieve real-time failure localization, we couple detection and localization closely and only rely on end-to-end measurements to localize failures without the help of other tools (*e.g.*, fbtracert [1]). To make it possible, we quantify several desirable properties of probe matrix (*e.g.*, identifiability) and propose a greedy algorithm to minimize probe cost. To address the scalability issue in DCNs, we apply several optimization heuristics and exploit characteristics of the DCN topology to accelerate probe path computation (§4).

▷ We modify a failure localization algorithm based on packet loss characteristics in large-scale data centers. Compared to the existing algorithms, our algorithm runs within seconds and achieves higher accuracy and lower false positive rate (§5).

▷ We implement and evaluate our system on a 4-ary Fattree tested built with 20 SDN switches. The experiments show that *deTector* can accurately localize failures in near real time with less probe overhead, *e.g.*, for 98% accuracy, *deTector* requires 3.9x and 1.9x times less probes than Pingmesh and NetNORAD while localizing failures 30 seconds in advance. Our supplementary simulation further shows that *deTector* achieves greater than 98% accuracy in failure localization with a less than 1% false positive ratio for most failures in large-scale DCNs (§6).

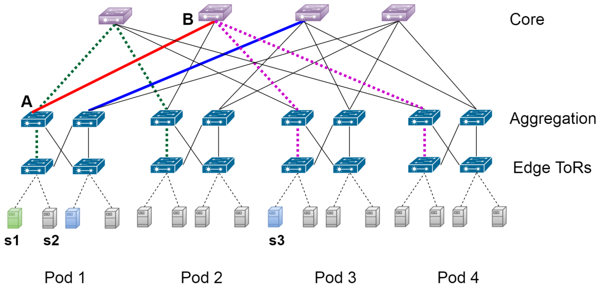


Figure 1: A 4-radix Fattree topology: failure on link *AB* can be detected by sending probes from $s1$ to $s3$.

## 2 Motivation

DCNs are usually multi-stage Clos networks with multiple paths between each pair of servers for load balancing and fault tolerance [4, 17, 21, 43]. Each DCN has its favorable routing protocols for path selection. For example, in a Fattree topology [4] and a VL2 topology [17], the shortest paths between any two ToRs are typically used in practice [25]. We describe how existing monitoring systems may fall short in achieving the three design objectives. Table 1 shows detailed comparison among *deTector*and the existing systems.

The passive approach stores packet statistics on switch counters, which are polled using SNMP or CLI periodically. In Fig. 1, if link *AB* is down, the switch counters will show a lot of packet losses. However, if the failure is a gray failure rather than link down, it may go undetected. For example, when silent packet drops occur (*e.g.*, in case of a packet blackhole that the switch silently drops packets with certain source and destination host addresses [21]), the switch may not provide any packet drop hint (*e.g.*, syslog errors) due to various reasons (*e.g.*, ASIC deficit), and hence SNMP data may not be fully trustworthy [21]. Furthermore, switch counter data can be noisy, such that problems identified by this approach may or may not lead to end-to-end delay or loss perceived by users.

Pingmesh [21] and NetNORAD [1] adopt an end-to-end probing approach to measure network latency and packet loss. Pingmesh selects probe paths by constructing two complete graphs within a DCN: one includes all servers under the same ToR switch (*i.e.*, the switch in the edge layer in Fig. 1) and the other spans all ToR switches. NetNORAD is similar to Pingmesh but places pingers in a few pods instead of all servers. Their approaches simplify the design but bring quite significant overhead (§6). Although gray failures can be captured, it is difficult to detect failures leading to low rate losses (*e.g.*, 1%) of a link, when ECMP is adopted in the DCN: there are many paths between a pair of servers, low-rate losses on a particular link may not affect much the overall end-to-end

Table 1: Comparison among *deTector* and existing representative monitoring systems

| | Gray failures | Low rate loss | Failure localization | Transient failures | Timeliness | Overhead |
|---|---|---|---|---|---|---|
| SNMP/CLI | No | No | Yes | Yes | minutes | switch resources |
| Pingmesh [21] | Yes | No | No, need Netbouncer | No | minutes | many probes |
| NetNORAD[1] | Yes | No | No, need fbtracert | No | minutes | many probes, switch CPU |
| *deTector* | Yes | Yes | Yes | Yes | near real-time | minimal probes |

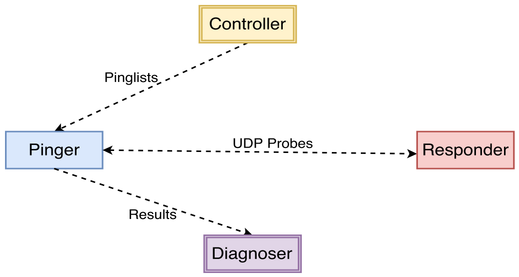


Figure 2: System architecture

loss rate between the two servers.

The exact location of losses cannot be pinpointed using Pingmesh or NetNORAD, since they do not know which paths the probes take (*e.g.*, due to ECMP). Therefore, other tools such as Netbouncer [2] and fbtracert [1] are needed, which send additional probes to play back the losses. These post-alarm tools may fail to pinpoint transient failures, those caused by transient bit errors, non-atomic rule updates or network upgrade (*e.g.*, a transient inconsistency between the link configuration and routing information [16]). To pinpoint such failures, close coupling of detection and localization is required, so that losses are localized only according to detection data, instead of additional probes after detection alarms. Such coupling further enables near real-time fault localization.

# 3 System Design

## 3.1 Architecture

*deTector* includes four loosely coupled components: a controller, a diagnoser, pingers and responders, as depicted in Fig. 2.

**Controller.** The logical controller periodically constructs the probe matrix indicating the paths for sending probes (§4). We mainly focus on failure localization on links inter-connecting switches, as the fault on a link connecting a server with a ToR switch can be easily identified as discussed in the next paragraph. The probe matrix indicates paths between ToRs. Since we do not rely on ToRs with pinging capability, probes are sent by 2–4 selected servers (pingers) under each ToR.

**Pinger.** Each pinger receives the pinglist from the controller, which contains server targets, probe format and ping configuration (§6.1). The probe paths from a ToR switch to different destinations are distributed among pinglists of pingers under the ToR switch, with each path distributed to at least 2 pingers for fault tolerance. In this way, in case that one pinger is down, other pingers in the same rack can still probe the paths, avoiding any large drop in link coverage. To detect failure on links connecting servers and the respective ToRs, pingers are also responsible for probing other servers under the same ToR. The number of probe paths for each pinger is no more than a hundred even for a large DCN (§4.4). The probe packets are sent over UDP, to minimize overhead from the perspective of operating the system. Though TCP is used to carry most traffic in a DCN, the DCN does not differentiate TCP and UDP traffic (*e.g.*, the forwarding behavior) in the vast majority of cases [34, 21], and hence UDP probes can also manifest the network performance. When a pinger detects a probe loss, it confirms the loss pattern by sending two probe packets of the same content additionally.

**Responder.** The responder is a lightweight module running on all servers. Upon receiving a probe packet, the responder echoes it back. A responder does not retain any states and all probing results are logged by pingers.

**Diagnoser.** Each pinger records packet loss information, and sends it to the diagnoser for loss localization. These logs are saved into a database for real-time analysis and later queries. The diagnoser runs the PLL algorithm (§5) to pinpoint faulty links and estimates the loss rates of suspected links.

For the controller and the diagnoser to be fault-tolerant and scalable, we can use existing solutions (*e.g.*, Software Load-Balancer [39, 21]).

## 3.2 Workflow Overview

*deTector* carries out three steps periodically: path computation, network probing and loss localization.

**Path computation.** At the beginning of each cycle, the controller reads the data center topology and server health from data center management service (*e.g.*, [27]), and selects the minimal number of probe paths (§4). The controller then selects pingers in each ToR, constructs and dispatches the pinglists to them.

**Network probing.** Next, probe packets are sent along the specified paths across the DCN. Since data center usually adopts ECMP for load balancing, we have to use source routing to control the path traveled by each probe packet, which can be implemented using various methods.[1] A general and feasible solution is to employ packet encapsulation and decapsulation to create end-to-end tunnels. Take the Fattree network in Fig. 1 as an example: fixing a core switch, there is only one path between two inter-pod servers; we can use IP-in-IP encapsulation to wrap a probe packet on a server; after the packet arrives at the core switch, the outer header is removed and the packet is routed to the real destination. Such a source routing mechanism incurs little overhead on servers and core switches.

**Loss localization.** The probe loss measurements are aggregated and analyzed by our loss localization algorithm (§5) on the diagnoser. We pinpoint the faulty links, estimate the loss rates, and send alerts to the network operator for further action.

## 4 Probe Matrix Design

The root cause of limitations of the existing monitoring systems is that the probe path selection is far from optimum, such that not enough useful information can be collected and additional probes are needed to reproduce losses for localization. In this section, we elaborate how we carefully select probe paths to overcome such a limitation.

### 4.1 Problem

Consider a data center network graph $G = (V,E)$, where $V$ is the set of switches and $E$ is the set of links. $\mathbf{R}$ is the $m \times n$ routing matrix defined by

$$\mathbf{R}_{i,j} = \begin{cases} 1 & \text{if link } j \text{ is on path } i \\ 0 & \text{otherwise} \end{cases}$$

where $m$ is the number of paths and $n = |E|$ is the number of links. The possible paths and the routing matrix are decided by the routing protocols employed in the data center, *e.g.*, ECMP is typically used to exploit $k^2/4$ parallel paths between any two ToRs in a $k$-ary Fattree. Fig. 3 gives a routing matrix $\mathbf{R}$ with 3 paths and 3 links. Note that each link in a DCN is typically bi-directional. Once we select a path from server $s1$ to server $s2$ and send a probe, the reverse path from $s2$ to $s1$ is automatically selected, since the response packet can probe faults along the reverse direction. When we identify that link $AB$ has failed, it implies that the failure may lie at either direction of the link, switch $A$, or switch $B$.

$$\mathbf{R} = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \end{array} \!\!\begin{array}{c} \begin{array}{ccc} l_1 & l_2 & l_3 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \rightarrow \mathbf{R'} = \begin{array}{c} \begin{array}{cccccc} l_1 & l_2 & l_3 & l_{12} & l_{13} & l_{23} \end{array} \\ \left(\begin{array}{ccc|ccc} 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{array}\right) \end{array}$$

Figure 3: Extend routing matrix with virtual links

**Problem 1** *Given a DCN routing matrix **R**, select a set of paths to construct a probe matrix **P**, such that **P** simultaneously (1) minimizes the number of paths, and achieves (2) $\alpha$-coverage and (3) $\beta$-identifiability.*

*Minimizing the number of probe paths* is desirable for minimizing network bandwidth consumption and analysis overhead, such that we may finish probing and diagnosing the entire DCN in merely a few minutes. Under the same probing bandwidth budget, it allows each pinger to probe the same set of paths more frequently.

*$\alpha$-coverage* requires that each link is covered by at least $\alpha$ paths in the probe matrix. Covering a link multiple times brings higher statistical accuracy for loss detection, as well as better resilience to pinger failures (since a link is more likely to be covered by probes from multiple pingers).

*$\beta$-identifiability* states that the simultaneous failures of any (no more than) $\beta$ links in the DCN can be localized correctly. For the routing matrix in Fig. 3, suppose we select $p_1$ and $p_2$ to constitute the probe matrix, *i.e.*, the probe matrix contains the first two rows of $\mathbf{R}$. If 2 or more links fail simultaneously, the faulty links cannot be correctly identified, as the observation from the end is the same, *i.e.*, packet losses are observed on both paths. On the other hand, if only one link is faulty, the bad link can be identified effectively: losses are observed on both paths, $p_1$, or $p_2$ if link 1, 2, or 3 is faulty, respectively. Therefore, the probe matrix achieves 1-identifiability, but not 2 or higher identifiability. Better identifiability contributes to higher accuracy of loss localization.

We find that Problem 1 is NP-hard for general DCNs as the Minimum Set Cover Problem is a special case of the problem. We hence resort to an approximation algorithm to compute the probing path.

### 4.2 PMC Algorithm

We extend a well-known greedy algorithm [8] for constructing a probe matrix achieving 1-identifiability to one achieving $\beta$-identifiability, as well as $\alpha$-coverage using a minimal number of probe paths.

In a probe matrix, a link belongs to a set of paths. To achieve 1-identifiability, the path sets of different links should all be different, so that losses can be observed on a particular set of paths to identify the faulty link. Recall that the set of links in our DCN is $E$. Once we select a path from the set of all feasible paths decided by the

[1] Source routing protocols have been designed in some DCNs like BCube [19] and DCell [20]; some work [25, 28] introduce other solutions for explicit path control.

routing matrix based on some criterion, it splits $E$ into two subsets $E_1$ and $E_2$, containing links on the selected path and the other links, respectively. If we do not observe any packet loss on this path, it implies that all links in $E_1$ are good; otherwise, there must be at least one bad link in $E_1$. Similarly, we select another path to further split $E_1$ and $E_2$ into smaller subsets, and repeat this procedure. Eventually if we can obtain subsets each containing only one link, then the probe matrix constructed using the selected paths achieves 1-identifiability (since the set of paths traversing each link is unique); otherwise, there does not exist a 1-identifiable probe matrix in the DCN. Throughout the process, if we always select a path whose links are present in the largest number of link sets to further split the link sets as much as possible, we will end up with the minimal number of paths needed.

To achieve $\beta$-identifiability, we expand the DCN graph $G$ with "virtual links". A virtual link is a combination of multiple physical links, and the set of paths a virtual link belongs to can be computed by "OR"-ing together the paths including the individual links [8]. For the example in Fig. 3, the original routing matrix $\mathbf{R}$ is extended to **R'** with three additional virtual links $l_{12}$, $l_{13}$ and $l_{23}$ added; the column corresponding to the virtual link $l_{12}$ can be computed by "OR"-ing the two columns corresponding to links $l_1$ and $l_2$. For $\beta$-identifiability, $\sum_{2\leq i\leq\beta} C(|E|,i)$ virtual links should be added in the DCN graph (routing matrix), corresponding to all combinations of 2 to $\beta$ links in the original graph. Then we can run the above algorithm for constructing 1-identifiable matrix based on the new routing matrix, and the resulting probe matrix achieves $\beta$-identifiability.

The probe matrix does not achieve even path coverage among the links yet. For example, for a 1-identifiable probe matrix constructed on a 64-ary Fattree, the gap between the maximal and minimal numbers of probing paths passing through any two links can be as large as 188. To achieve better evenness (*i.e.*, spreading paths and thus probe overhead evenly among the physical links), we introduce a link weight $w[link]$, denoting the number of paths that the link resides on, and ensure that it is no smaller than $\alpha$ for any physical link. We also define a score for each (extended) path, *i.e.*, the path includes virtual links from the extended routing matrix **R'**:

$$score(path) = \sum_{link\in path} w[link] - \#\text{ of link sets on } path \tag{1}$$

Here the link sets are the split link sets produced by the procedure above. We say that a link set is on a path if the link set contains at least one (physical or virtual) link of the path. Thus, a lower score indicates that the links on the path are not covered much by paths already selected and/or more link sets can be split if the path is selected in the above procedure. We strive to achieve better evenness among the links while guaranteeing $\alpha$-coverage, by always selecting a path with the lowest score.

**Algorithm 1** Probe Matrix Construction (PMC) Algorithm

```
Require: $\mathbf{R}$, $\alpha$, $\beta$
 1: Initialize $w$, $score$ to $\mathbf{0}$, $setnum$ to 1, $selpaths$ to $\emptyset$
 2: $\mathbf{R'} \leftarrow LINKOR(\mathbf{R}, \beta)$
 3: $paths \leftarrow$ all paths in $\mathbf{R'}$, $physlinks \leftarrow E$
 4: while ($setnum \neq |E|$ || $physlinks \neq \emptyset$) && $paths \neq \emptyset$ do
 5:     for $path \in paths$ do
 6:         update $score[path]$ according to (1)
 7:     $path \leftarrow \mathrm{argmin}_{path' \in paths}\, score[path']$
 8:     $selpaths \leftarrow selpaths \cup \{path\}$
 9:     $paths \leftarrow paths/\{path\}$
10:     for $physlink$ on $path$ do
11:         $w[physlink] \leftarrow w[physlink] + 1$
12:         if $w[physlink] > \alpha$ then
13:             $physlinks \leftarrow physlinks/\{physlink\}$
14:     update $setnum$ as the total number of link sets after split by $path$
15: return probe matrix constructed by paths in $selpaths$ (retaining only physical links on the paths)
```

Our Probe Matrix Construction algorithm, PMC, is summarized in Alg. 1. We first reduce the problem of constructing a $\beta$-identifiable matrix to one constructing 1-identifiable matrix, by adding virtual links to the original routing matrix of the DCN graph (line 2, where *LINKOR* denotes the method for extending routing matrix discussed above). Then in each iteration we update the score of each (extended) path (lines 5-6) and select a path which has the *minimal* score among all candidate paths (lines 7-8). We remove the selected path from the candidate path set (line 9), and update the weight of physical links ($w[physlink]$) on the selected path (lines 10-11) and the total number of link sets that the already selected paths can split into (line 14, which corresponds to the procedure discussed in the second paragraph of this subsection). If the number of paths that cover one (physical) link exceeds $\alpha$, we remove the link from the set of all links (line 12-13). The loop stops when the probe matrix achieves $\alpha$-coverage (i.e., the set *physlinks* is empty) and $\beta$-identifiability (i.e., the number of link sets split equals the number of links), or there are no more candidate paths (*i.e.*, the set *paths* is empty).

**Theorem 1** *The PMC algorithm achieves* $(1-\frac{1}{e})$ *approximation of the optimum in terms of the total number of probe paths selected.*

We can prove Theorem 1 by showing that the score of a path set is monotone, submodular and non-negative.

The detailed proof is in Appendix A. In practice, the PMC algorithm performs much better than the $(1-\frac{1}{e}) \approx 0.63$ approximation ratio (§4.4). The issue of this algorithm, however, is the computation time. The time complexity of the algorithm is $O(m^2)$, where $m$ is the number of paths, since in the worst case we may update the scores of all paths in each iteration and end up with selecting all paths. In a 64-radix Fattree, there are about $4.3 \times 10^9$ desirable paths among ToRs. As we will see in §4.4, the algorithm is still too slow for any data center at a reasonable scale, and we adopt a number of optimizations to further speed it up.

## 4.3 Algorithm Speedup

To speed up the PMC algorithm, we apply several optimizations based on the following three observations.

**Observation 1** *Problem 1 can be divided into a series of subproblems.*

We can construct a bipartite graph according to the routing matrix: one partition corresponds to paths and the other consists of links; an edge exists between a path node and a link node if the link is on the path. We observe that if the routing matrix can be partitioned into sets of paths with no links in common, then the problem can be divided into independent subproblems. For example, in Fig. 1, paths traversing the red link have no link overlapping with paths traversing the blue link. Therefore, the bipartite graph can typically be divided into connected subgraphs and each subgraph represents a smaller routing matrix and hence a subproblem. Finding connected subgraphs can be done in linear time by traversing the bipartite graph once. Then the PMC algorithm can be applied to the subproblems in parallel.

**Observation 2** *Not all paths require score update in each iteration.*

Once we select a path, only the scores of those paths which share at least one link with the selected path change. Therefore we only need to update these paths' scores in lines 5-6 of Alg. 1. We call this *partial update*.

**Observation 3** *The score of each path is non-decreasing over all iterations.*

It can be proved that the score of a path is non-decreasing (Appendix A). Inspired by the CELF algorithm for outbreak detection in networks [35], we adopt a strategy called *lazy update* which defers the update of a path score as much as possible even though we know the score is outdated. Specifically, we maintain a min-heap for all paths with scores as the keys and only update the score of a path when the path is at the top of the heap. After score update, if the path still stays at the top of the heap, *i.e.*, the path has the minimal score among all available paths, we will select the path as a probe path, even though some path scores have yet to be updated. The correctness of this heuristic is guaranteed by submodularity of the score of a path set: the marginal gain provided by a path selected in the current iteration can not be larger than that provided by the path in the previous iteration.

**Observation 4** *The DCN topology is typically symmetric.*

Due to symmetry, when a path is selected, all its topologically isomorphic paths can be selected. For example in Fig. 1, if the dashed green path spanning Pod 1 and Pod 2 is selected, then the dashed purple path spanning Pod 3 and Pod 4 may be a good choice too. This helps us reduce the scale of the problem since the routing matrix **R** can be reduced to a smaller matrix by excluding paths that are topologically isomorphic to other paths. For example, if the green path is in the matrix, we do not need to include the purple path. For this purpose, we first need to compute the symmetric components in a DCN graph. There are many fast algorithms available for symmetry discovery [12, 10], *e.g.*, $O_2$ [10] can finish computation within 5 seconds for a Fattree(100) DCN, and we only need to precompute it once for a DCN.

## 4.4 Performance

We run our PMC algorithm on a Dell PowerEdge R430 rack server with 10 Intel Xeon E5-2650 CPUs and 48GB memory, to test its running time and number of paths selected. We compare results on three well-known DCNs, Fattree [4], VL2 [17] and BCube [19].[2]

**Running time.** Table 2 shows the algorithm running time for constructing a probe matrix achieving 2-coverage and 1-identifiability. The strawman approach is our PMC algorithm without any optimizations. The last four columns contain results when the respective optimization is in place (in addition to the previous one(s)). The results show that PMC can efficiently select probe paths for very large DCNs. Specifically, without algorithm speedup, the computation time of PMC can be larger than 24 hours; after each optimization, the time decreases significantly and we can compute the probe matrix for Fattree(72), VL2(140,120,100) and BCube(8,4) within 18 seconds, 86 seconds and 70 seconds, respectively. We note that the running time in case of problem decomposition for VL2 and BCube is a bit longer than that of strawman. This is because decomposition does not apply to the two DCN topologies, but we need extra time to decide whether the matrix is decomposable.

[2] BCube is a server centric architecture and we treat servers as switches to run our algorithm.

Table 2: Algorithm running time (seconds) with $\alpha = 2, \beta = 1$ in different DCNs

| DCNs | # of nodes | # of links | # of original paths | Strawman | Decomposition | Partial update | Lazy update | Symmetry |
|---|---|---|---|---|---|---|---|---|
| Fattree(12) | 612 | 1296 | 184,032 | 231.458 | 5.216 | 1.509 | 0.506 | 0.126 |
| Fattree(24) | 4,176 | 10,368 | 11,902,464 | > 24h | 1381.226 | 99.778 | 23.254 | 0.280 |
| Fattree(72) | 99,792 | 279,936 | 8,703,770,112 | > 24h | > 24h | > 24h | > 24h | 17.054 |
| VL2(20, 12, 20) | 1,282 | 1,440 | 70,800 | 22.030 | 23.126 | 3.274 | 0.77 | 0.253 |
| VL2(40, 24, 40) | 9,884 | 10,560 | 4,588,800 | 7387.412 | 7470.476 | 385.287 | 39.028 | 1.404 |
| VL2(140,120,100) | 424,390 | 436,800 | 4,938,024,000 | > 24h | >24h | >24h | >24h | 85.567 |
| BCube(4, 2) | 112 | 192 | 12,096 | 4.871 | 4.936 | 1.229 | 0.227 | 0.117 |
| BCube(8, 2) | 704 | 1,536 | 784,896 | 4050.776 | 4390.168 | 164.345 | 9.854 | 0.220 |
| BCube(8, 4) | 53,248 | 163,840 | 5,368,545,280 | > 24h | > 24h | > 24h | > 24h | 69.778 |

Table 3: Number of selected paths with different ($\alpha$,$\beta$)

| DCNs | Original paths | Selected paths with ($\alpha$,$\beta$) | | |
|---|---|---|---|---|
| | | (1, 0) | (1, 1) | (3, 2) |
| Fattree(32) | 66,977,792 | 4,096 | 7,680 | 12,288 |
| Fattree(64) | 4,292,870,144 | 32,768 | 61,440 | 98,304 |
| VL2(72,48,40) | 107,371,008 | 864 | 1,440 | 2,640 |
| VL2(128,96,80) | 2,415,132,672 | 3,072 | 5,760 | 9,216 |
| BCube (8,2) | 784,896 | 1,712 | 2,016 | 2,832 |
| BCube (8,4) | 5,368,545,280 | 49,152 | 70,572 | 119,556 |

**Path number.** Table 3 shows the number of selected paths with different $\alpha$ and $\beta$ in different DCNs. Compared with the number of original paths in DCNs, PMC only selects a small percentage of paths. We can prove that the least number of paths for achieving 1-coverage and 1-identifiability is $k^3/5$ for any $k$-ary Fattree (Appendix B). Thus, a Fattree(64) DCN needs at least 52428 paths and our algorithm selects slightly more, *i.e.*, 61440 paths. This implies that pingers under each selected ToR in the Fattree are only responsible for probing about 60 paths, much fewer than that of Pingmesh (about 2000-5000 paths). We also find that VL2 requires much fewer paths than Fattree and BCube. This is because VL2 has a much smaller number of links between switches (12288 links in VL2(128, 96,80)), as compared to Fattree (131072 links in Fattree(64)) and BCube (163840 links in BCube(8,4)).

Note that the number of selected path may change when the fourth optimization, based on topology *symmetry*, is in place. Our evaluation shows that the number of selected paths with symmetry reduction is very similar to that without symmetry reduction. This is consistent with the result in [25], and we hence omit the analysis.

**Results for $\beta \geq 3$.** The probe matrices we constructed above achieve at most 2-identifiability. For $\beta \geq 3$, the computation of PMC is not efficient in large DCNs. For the example of a 48-ary Fattree, we can compute a probe matrix achieving at most 2-identifiability in a reasonable amount of time (*i.e.*, within 10 mins), while 3-identifiability requires at least 24 hours, even when we apply all speedup optimizations in §4.3. The fundamental reason is that the routing matrix $\mathbf{R}$ becomes much larger when the number of column increases from $n$ to $\sum_{1\leq i\leq\beta} C(n,i)$, by adding virtual links. However, surprisingly, we find that 2-identifiability is enough for loss localization in DCNs, as we will see in §6.4.

## 5 Loss Localization

Every half a minute, a pinger collects packet loss information on each path it probes, and sends the loss information to the diagnoser (see details in §6.1). The diagnoser periodically analyzes the collected data to localize failures.

### 5.1 Data Pre-processing

The first step is to pre-process the collected loss data, to remove outliers and normal cases. Severe packet losses could be caused by bad pingers and responders (*e.g.*, the server is down or was rebooting during probing, thus causing many false alarms [34]). Such outliers can be identified by keeping track of the status of servers used as pingers and responders. In addition, a link normally has a regular low loss rate, *e.g.*, $10^{-4}$–$10^{-5}$, due to background noise such as network congestion, increased server load, bit error rate, which should not be considered as failures [21]. To exclude such normal cases, we filter out paths with extremely low packet loss. While setting a threshold on the number of packet losses in a period or on path loss ratio (*e.g.*, $10^{-3}$ [21]) is a possible solution, it may filter out partial failures that cause low loss rates (§5.2). Thus, we use statistical hypothesis testing to look at loss rates over time to filter noisy data, which has been shown to be a good solution to produce a robust signal [22]. After pre-processing, the loss data that remain are likely manifest of network failures rather than noises.

### 5.2 Problem

Let $\mathbf{Y}$ denote the vector that contains packet loss records ($path$, $packet\ losses$), for all selected paths in the probe matrix (excluding those paths removed during data pre-processing).

**Problem 2** *Given an end-to-end packet loss observation* $\mathbf{Y}$*, find the smallest set of faulty links that best explains the observation.*

The fault localization Problem 2 is NP-hard as it can be reduced to the NP-complete Minimum Hitting Set Problem [13]. Before we present our heuristic algorithm, we first describe the differences between our problem at hand and failure (loss) localization problems dealt with in existing work (*e.g.*, [14, 31, 5, 29]).

*Much larger problem scale.* Classical work (*e.g.*, [5, 13]) on fault localization treat failures as exceptions rather than norm, *i.e.*, the number of failures is very limited (usually at most 3). Hence the running time of the existing algorithms can be satisfactory. However, packet loss in a large-scale data center is quite common (*e.g.*, thousands of paths with packet loss in a large DCN simultaneously due to network update) and our network scale is much larger. At our problem scale, the existing algorithms are not fast enough (taking tens of seconds or even minutes) for real-time loss localization.

*Different loss patterns.* While there are many kinds of network failures, they are exhibited as two kinds of packet losses: full packet loss and partial packet loss, meaning that all or part of the packets traversing a link are dropped. Full loss can be triggered by switch down, line card not seated well, etc; partial packet loss can happen due to bit flips, ASIC deficits, etc [21]. Boolean tomography [13, 26] is the state-of-the-art technique for localizing the most likely failed links among previous studies on loss tomography in general Internet settings. It assumes that if all links on a path are good, then the path is good [14]. But this assumption is not true in case of partial packet loss in data centers, *e.g.*, packet blackhole may cause a link to fail only for a subset of paths using that link. To improve the diagnosis, control plane information from switches is used in some work [13, 22]. Since *deTector* treats each switch as a blackbox rather than whitebox (§1), we have to resort to other solutions.

## 5.3 PLL Algorithm

We design an efficient method to localize both full and partial packet losses. Our approach is based on a greedy algorithm *Tomo* in [13]. The basic idea of *Tomo* is as follows: We first exclude those links on working paths (i.e., the paths without any loss); then in each iteration we compute a score for each link as the number of lost packets that the link can explain.[3] We greedily select the link with the maximal score and remove those losses this link can explain. This procedure repeats until no loss remains unexplained.

**Algorithm 2** *PLL*: Packet Loss localization Algorithm

```
Require: packet loss measurements $\mathbf{Y}$, probe matrix
decompose Problem 2 and run the following procedure for each subproblem (where $\mathbf{Y}$ contains loss measurements of paths in the subproblem) in parallel
procedure
    divide $\mathbf{Y}$ into two vectors $\mathbf{Y}_f$ and $\mathbf{Y}_p$
    $badlinks \leftarrow Tomo(\mathbf{Y}_f)$
    $badlinks_p \leftarrow Tomo'(\mathbf{Y}_p)$
    for $link \in badlinks_p$ do
        compute $hitratio[link]$
        if $hitratio[link] > thrsh$ then
            $badlinks \leftarrow badlinks \cup \{link\}$
    Compute link loss rates for all links in $badlinks$
    return all $badlinks$ and their loss rates
```

*Tomo* relies on boolean tomography and can not handle partial link loss itself, and hence it uses router logs to narrow down the set of potentially failed links. Our algorithm differs from *Tomo* in dealing with partial link losses. In our localization method, we divide the packet loss vector $\mathbf{Y}$ into two vectors: $\mathbf{Y}_f$ and $\mathbf{Y}_p$, containing paths (with the number of packet losses) where full packet loss occurs and paths (with the number of packet losses) with partial packet losses, respectively. This can be done by checking whether all probe paths that a link lies on suffer packet loss. If so, move these paths to $\mathbf{Y}_f$. The rest paths will be included in $\mathbf{Y}_p$. *Tomo* is applied directly on $\mathbf{Y}_f$ to identify faulty links. For paths with partial failures in $\mathbf{Y}_p$, we apply *Tomo* but make two improvements: (1) we do not use working paths to exclude possibly bad links because a link with partial link loss may manifest as good for some probe paths traversing it; (2) we only output a bad link if its hit ratio is larger than a threshold. The hit ratio of a link is the ratio of the number of observed failed paths in $\mathbf{Y}_p$ through the link over the number of all probe paths through the link [30]. For example, if four out of five probe paths covering the same link experience packet losses, the hit ratio of the link is 0.8. The hit ratio quantifies how "partial" the packet loss is on a link. Note that there is a tradeoff in setting the threshold for hit ratio (§5.4): a larger threshold means less false positives (*i.e.*, good links incorrectly identified as bad) but may lead to more *false negatives* (*i.e.*, bad links incorrectly identified as good). Compared to Tomo, our algorithm gives operators more freedom to adjust the algorithm performance according to their needs.

Our Packet Loss Localization algorithm, *PLL*, is given in Alg. 2. We first divide Problem 2 into a series of subproblems, by decomposing the probe matrix (with some paths excluded during data pre-processing) following the same steps discussed for decomposing the routing matrix in §4.3, *i.e.*, paths in $\mathbf{Y}$ are divided among the subprob-

[3] If a link lies in the packet path, we say the link can explain the packet loss.

Table 4: Fault localization accuracy and running time in a 48-radix Fattree

| # of failed links / Algorithms | Accuracy (%) | | | | False positive ratio (%) | | | | Running time (seconds) | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 5 | 20 | 50 | 1 | 5 | 20 | 50 | 5 | 50 |
| *Sherlock* [5] | 33.90 | 5.57 | 1.45 | 1.24 | 10.26 | 14.43 | 34.58 | 56.10 | 79.200 | 225.600 |
| *SCORE* [30] | 93.86 | 90.27 | 90.02 | 88.77 | 3.68 | 4.37 | 4.42 | 5.19 | 0.720 | 9.600 |
| *OMP* [40] | 89.74 | 91.38 | 49.43 | 19.37 | 6.82 | 11.61 | 23.22 | 44.62 | 14.400 | 16.320 |
| *Tomo* [13] | 94.48 | 90.45 | 89.88 | 88.87 | 3.74 | 4.61 | 4.51 | 5.31 | 0.672 | 9.360 |
| *PLL* | 96.00 | 91.87 | 91.24 | 89.87 | 1.73 | 2.95 | 3.18 | 3.84 | 0.486 | 0.772 |

lems. Each subproblem can be handled independently in parallel (line 1), to accelerate computation. We use *Tomo* to derive faulty links on paths in $\mathbf{Y}_f$ (line 4), and use the hit ratio to filter output links when applying $\mathit{Tomo}'$ on $\mathbf{Y}_p$ (lines 5-9, $\mathit{Tomo}'$ is a modified version of *Tomo* with our first improvement (1) above). We combine bad links derived from $\mathbf{Y}_f$ and $\mathbf{Y}_p$ and compute the link loss rates on them (*i.e.*, the number of lost packets explained by the link divided by the number of all packets sent through the link) (line 10). Faulty links obtained in all the subproblems together with their loss rates are sent to the network operator of the DCN for further action (*e.g.*, taking the links down or repairing the links).

## 5.4 Performance

We generate network failures based on the measurement data in [7] (see §6.2 for more details) and run the PLL algorithm on our Dell server to test its fault localization accuracy, false positives and running time. The accuracy of failure localization is defined as the true positive ratio as in many other work [31, 44], *i.e.*, the percentage of bad links correctly identified as bad over all truly bad links. Since high accuracy means low false negatives, we do not present the result of false negatives.

Fig. 4 shows the accuracy and the false positive ratio (*i.e.*, the percentage of false positives over all correctly and incorrectly identified links) when pinpointing 3 link failures in a 48-radix Fattree with a 1-identifiable probe matrix, which are randomly selected full or partial failures following the model in §6. With the increase of the hit ratio threshold, both the accuracy and the false positive percentage experience a sudden drop. That is, based on link loss characteristics from the measurement data we used in [7], the hit ratio is bimodal and we are more likely to see that a link either drops all packets on it (*i.e.*, high hit ratio) or only drops packets on a few certain paths (*i.e.*, low hit ratio). Note that the result may not be the same in DCNs with different traffic and link characteristics, and thus setting the threshold requires network operator's experience and, if possible, by learning from real loss data. According to Fig. 4, the false positive ratio decreases a lot (from 5.5% to 1.3%) while the accuracy still remains higher than 90%. Hence we suggest a relatively higher hit ratio threshold to keep the false positives low (we set it to be 0.6 by default in the following simulations and experiments).

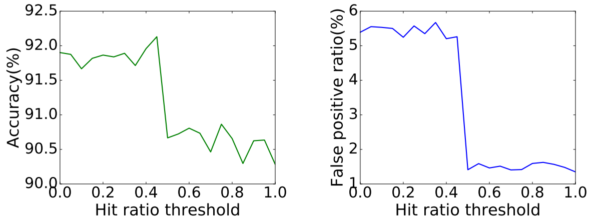


Figure 4: Accuracy of PLL to localize 3 failures in a 48-radix Fattree

Table 4 shows a comparison of PLL with other failure localization algorithms, *Sherlock* [5], *SCORE* [30], *OMP* [40] and *Tomo* [13], when localizing different numbers of failures using a 1-identifiable probe matrix in a 48-ary Fattree. With the increase of the number of failed links, the accuracy of all algorithms decreases (*i.e.*, the false negative increases), but *PLL* always performs the best. *Sherlock* defines an explainability for each link,[4] but the explainability does not accurately quantify how likely a link is bad since a high true positive ratio does not imply a low false positive ratio as shown in Fig. 4. *SCORE* is a general failure root causing algorithm in backbone networks. It does not distinguish between full packet loss and partial packet loss when applied for loss localization, and thus it does not handle partial link loss well, just as *Tomo*. *OMP* is widely used in compressed sensing to compute representations of signals; its method for updating loss measurements after selecting a faulty link is not suitable for packet losses since this method tends to attribute the packet losses on a path to multiple faulty links, which seldom occurs in practice. Note that *PLL* also achieves a lower false positive ratio than the other four algorithms due to the introduction of hit ratio threshold. Though based on *Tomo*, *PLL* runs much faster due to problem decomposition, *e.g.*, it can finish failure localization within 1 second in a large DCN with 82944 links.

[4] explainability=TP/(TP+FN+FP) where TP, FN, FP represent true positives, false negatives and false positives respectively.

# 6 Implementation and Evaluation

## 6.1 Implementation

We run the controller on one Dell server (or it can run in a distributed fashion over multiple servers for large-scale networks). A watchdog service also runs on the server for monitoring the health of other servers and removing bad ones. The controller runs the PMC algorithm to recompute the probe matrix every 10 minutes, based on the current network topology from the watchdog service.[5] The computed probe matrix is divided into XML pinglist files for dispatching to pingers. A pinglist file contains file version, the pinger's IP address, IP addresses of responders, transport port numbers, the packet-sending interval and IP addresses of core switches. Our measurement shows that the controller can handle 4473 pinglist requests per second on average with maximal bandwidth consumption 688.56Mb/s using only one core. Since pingers are deployed on a small number of servers (about 10% among all servers) and fetch pinglists in each cycle, the controller can support more than 100,000 pingers by slightly randomizing the time when pingers request for pinglists.

Each pinger implements a communication module and a probing module. The communication module is responsible for connections with the controller and the diagnoser. It fetches the pinglist file from the controller by a HTTP GET request in every cycle (*i.e.*, 10 minutes). The probing module generates probe packets according to the pinglist and encapsulates them by IP-in-IP (§3.1). In our experiments, a pinger loops over a range of ports for each path, and emits several packets for every port. Each probe packet has an average size of 850 Bytes, carrying a specified DSCP value in the IP header to test different QoS classes in DCNs [7]. If there is no response for a probe within 100ms, we mark it as a loss. A pinger repeatedly sends packets by looping through the paths in the pinglist for multiple times (for statistical accuracy), at the rate of 10 packets per second. Every 30 seconds, the pinger aggregates the probing results (*i.e.*, the number of packet losses and the number of total packets sent on each probe path) into an XML file and sends it to the diagnoser by a HTTP POST request. The responder module runs on the userspace of all servers, which listens to a particular port, and upon packet arrival, it adds a timestamp and sends the packet back. The pinger and responder incur little overhead on servers, as we will see in §6.3.

The diagnoser is a Web server module, which runs on the same server where the controller is in our experiments, and processes the POST data for fault localization. It runs the PLL algorithm once every half a minute, using collected probe results in the past 30 seconds. Given the limited number of servers in our testbed, we run a virtual machine to emulate a server in our experiments. We have open sourced *deTector* [3].

## 6.2 Experiment Setup

We build a 4-ary Fattree testbed with 20 SDN hardware switches [24, 23], each equipped with four 1GbE ports and one dedicated management port. Though we do not require programmable switches in *deTector*, employing SDN switches facilitates our emulation of various failure cases that may happen in a real-world DCN. Specifically, we categorize all losses into three types:

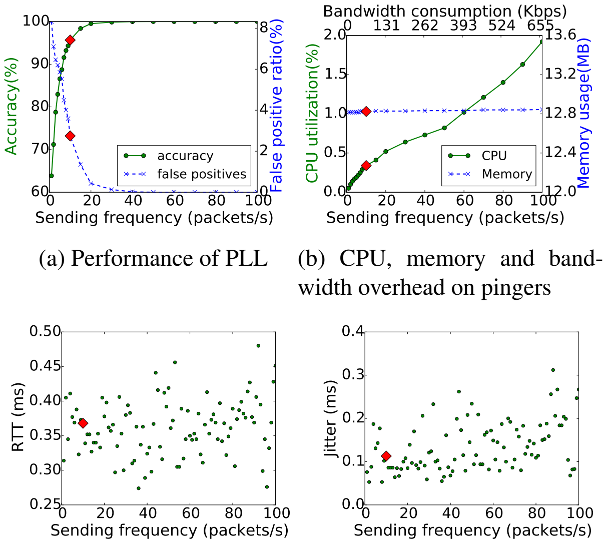
**Full packet loss.** We install OpenFlow rules with high priority to drop all packets coming from a particular port, to emulate a faulty link with full packet loss. To emulate a switch down case, we install rules to drop all packets at the switch.

**Deterministic partial loss.** Packets with certain features (*e.g.*, specific IPs, port numbers) may be dropped on a link deterministically, *e.g.*, in case of packet blackhole or misconfigured routing rules. To emulate such failures, we install rules on the switches to match and drop packets with certain headers.

**Random partial loss.** Sometimes packets on a link are dropped randomly, as caused by bit flips, CRC errors, buffer overflow, etc. SDN switches do not support random packet dropping. To emulate such losses, we install rules on the switches to redirect all packets on an emulated bad link to the SDN controller, and the SDN controller drops the received packets with certain probability, following the pattern extracted from [7].

Due to no access to loss data in real-world data centers, we produce the above loss types according to the failure measurements in [15] and traffic measurements in [7]. Specifically, we set parameters such as link vs. switch failure percentage, link loss rates (ranging from $10^{-4}$ to 1), failure probabilities for switches in different tiers, all based on the above measurements. The loss distribution for links in different tiers is extracted from Fig. 3 in [7]. Aside from *deTector*, we also implement the probing modules of Pingmesh and NetNORAD on our testbed for performance comparison, as well as their failure localization tools, Netbouncer and fbtracert. Since we do not know some of their implementation details (*e.g.*, how data pre-processing is done), we implement those details in the same way across all three systems.

[5] Once a link or a switch has failed, we remove related link(s) from the routing matrix to avoid selecting bad paths for probing. Note that it does not affect symmetry computation which only pre-runs once on the original DCN topology.



(a) Performance of PLL

(b) CPU, memory and bandwidth overhead on pingers

(c) RTT of workload traffic

(d) Jitter of workload traffic

Figure 5: Sensitivity test of sending frequency

## 6.3 Performance

We first investigate how probing itself affects the whole DCN. We use realistic packet traces (including information such as packet header, timestamp) from a production data center [6] to generate workload traffic in our testbed, where each server continuously replays flows based on the packet traces and sends them to a random receiver. We evaluate how our probing frequency (*i.e.*, the number of probes a pinger sends per second) affects the performance of the PLL algorithm, the overhead on the pinger, and RTT and jitter experienced by the workload traffic. In each minute of our experiment, we emulate a failure randomly picked among the above three types of failures, with the failed switches or links randomly picked in the DCN. We run our experiment for 2000 minutes and obtain the average results.

Fig. 5 shows that a higher probe sending frequency leads to a higher accuracy and a lower false positive ratio, but causes higher CPU utilization and bandwidth consumption on pingers as well as slightly larger fluctuation of the RTT and jitter experienced by the workload. We find that 10–15 probes per second is good enough since we can still achieve higher than 95% accuracy and a lower than 3% false positive ratio, while only consuming about 100Kbps bandwidth, 0.4% CPU and 13MB memory on each pinger. Besides, it does not introduce apparent delay and jitter variations for workoad traffic. Note that the overhead of a responder is much smaller than a pinger because it resumes fewer tasks (*e.g.*, no communication with the controller and the diagnoser), and hence the results are omitted.[6] In all our experiments, the pinger sends 10 packets per second by default (*i.e.*, the red square in Fig. 5).

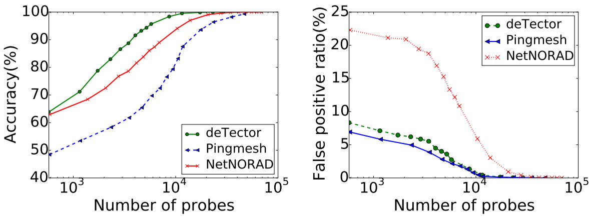


Figure 6: Accuracy and false positives of three monitoring systems with different number of probes per minute

We then compare the accuracy, false positive ratio and overhead among *deTector*, Pingmesh and NetNORAD. Since Pingmesh can not localize failures by itself, once it detects a suspected source-destination server pair, we use Netbouncer [2] to go through all possible paths between this server pair for loss localization. As for NetNORAD, similarly, we use fbtracert [1] to probe all possible paths between the suspected server pair. The interval of loss data collection is 30 seconds for three systems.

Fig. 6 shows the comparison when one failure is emulated in the testbed (the failure is randomly picked as in the previous experiment). The number of (ping and reply) probes in the figure includes probes sent for detection and probes for localization (if any) in each minute of the experiment. More probes indicate not only more bandwidth consumption, but also higher CPU and memory usage. For *deTector*, we use a probe matrix with 1-identifiability and 3-coverage (since it is impossible to achieve 2-identifiability in a 4-ary Fattree). As we can see, *deTector* achieves high accuracy and a low false positive ratio with a much smaller number of probes, because *deTector* covers more types of losses (*e.g.*, low rate loss) and takes carefully planned paths. For instance, to achieve 98% accuracy and 1% false positives, *deTector*, NetNORAD and Pingmesh need to send 7200, 20700 and 35100 probes per minute, respectively. When the probe overhead is same (same number of probes sent per minute), the accuracy and false positive ratio achieved by *deTector* is better than those of NetNROAD; as compared to Pingmesh, the accuracy of *deTector* is much better, while the false positive ratio of Pingmesh is slightly smaller sometimes, since it possibly probes all paths.

Fig. 7 further shows the accuracy and false positive ratio with multiple failures, when the probe overhead is fixed to be the same, *i.e.*, 5850 probes per minute. *deTector* always achieves much better performance than Pingmesh and NetNORAD. Note that *deTector* also detects and localizes failures much faster than NetNORAD

[6] Even when we place the pinger and responder on the same server, the overhead is negligible.

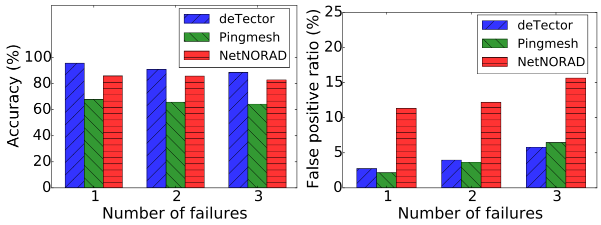


Figure 7: Results comparison with multiple failures

Table 5: Accuracy in a 18-radix Fattree, with probe matrices of different levels of coverage and identifiability

| $(\alpha,\beta)$ | # of paths | Accuracy (%) with # of failed links | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 5 | 10 | 20 | 50 |
| (1,0) | 729 | 30.56 | 30.87 | 30.30 | 30.26 | 29.19 |
| (2,0) | 1485 | 58.43 | 57.43 | 57.08 | 56.81 | 57.11 |
| (3,0) | 2187 | 68.22 | 70.61 | 69.89 | 70.40 | 70.14 |
| (1,1) | 1269 | 94.74 | 93.37 | 94.21 | 93.43 | 90.29 |
| (1,2) | 1512 | 99.26 | 99.06 | 99.02 | 98.77 | 95.92 |
| (1,3) | 2349 | 99.63 | 99.63 | 99.67 | 99.62 | 98.07 |

and Pingmesh (30 seconds in advance in our experiments), because *deTector* does not need any other diagnosis tools to send an additional round of probes for loss localization, while others do.

## 6.4 Simulation

We supplement our experimental evaluation with simulations, to investigate how identifiability of the probe matrix influences the accuracy of our failure localization, when running *deTector* in larger Fattree networks.

We first vary $\alpha$ and $\beta$ for probe matrix construction in a 18-radix Fattree. Table 5 shows that higher coverage and higher identifiability lead to higher accuracy, while the overhead (*i.e.*, the number of selected paths) does not increase much. Also, we find that identifiability is more effective and desirable than coverage for failure localization, since a 1-identifiability matrix increases the accuracy a lot (from one with 0-identifiability guarantee), with much less overhead than a 3-coverage probe matrix.

Note that further increasing the level of identifiability for $\beta > 1$ does not increase the accuracy much, and probe matrices achieving 1-identifiability can already lead to higher than 90% accuracy. According to the measurements in [7], less than 10% failure events (failures occurring concurrently) contain more than four failures and less than 1% failure events contain more than 20 failures. This implies that a probe matrix with 1-identifiability can guarantee higher than 93% accuracy for 90% failure events and 2-identifiability provides a 98% accuracy for 99% failure events.

The result is surprising but reasonable: Since we use a number of optimizations (§4.3) to reduce the size of the routing matrix, the PMC algorithm in fact achieves $\beta'$-identifiability (where $\beta'$ is larger than $\beta$ used in the algorithm) for the whole probe matrix, rather than $\beta$-identifiability computed for each small probe matrix (corresponding to a small network topology). Therefore, *deTector* may fail to localize all failures only if more than $\beta$ failures appear in a small topology, which occurs with relatively low probability. This shows that using a probe matrix with a low level of identifiability guarantee is good enough to identify a much larger number of concurrent failures.

Table 6: Fault localization performance with probe matrix of 2-identifiability in a 48-ary Fattree

| # of failed links | 1 | 5 | 10 | 20 | 50 |
|---|---|---|---|---|---|
| Accuracy (%) | 98.95 | 98.99 | 98.98 | 98.93 | 98.87 |
| False positive (%) | 0.01 | 0.02 | 0.02 | 0.02 | 0.02 |
| False negative (%) | 1.05 | 1.01 | 1.02 | 1.07 | 1.13 |

In addition, by examining the failure events that *deTector* fails to localize with a low identifiability probe matrix but can identify using a high identifiability matrix, we find that higher identifiability achieves better results only when the number of simultaneously failed links is very large. Such a failure event with many concurrent link failures is usually triggered by a common bug in practice (*e.g.*, 180 links fail simultaneously due to scheduled maintenance to multiple aggregation switches [15]), and thus those faulty links are spatially clustered. In such cases, operators can locate the failure spot effectively according to the positions of most failed links.

We further examine the fault localization accuracy, false positive and false negative (bad links incorrectly identified as good) ratios achieved using a probe matrix of 2-identifiability in a 48-ary Fattree. Table 6 shows that the false positive and false negative ratios remain in a very low level. In particular, the false positive rate is extremely low (< 1%), which is desirable in practice [13].

The false negatives are mainly caused by losses of extremely low loss probability and intermittent losses which may happen at longer intervals (than 1 minute) [18]. Since it takes longer time to expose these losses, we can further reduce false negatives by examining loss measurements in larger time windows, *e.g.*, 10 minutes or 30 minutes.

## 7 Discussions

**Packet entropy.** *deTector* tries to increase packet entropy (*i.e.*, different packet patterns) by varying IP addresses, port numbers and DSCP values, to cover as many failures as possible. However, (1) our implementation uses IP-in-IP encapsulation for source routing, hence the range of destination IP addresses is somewhat limited; (2) since we use a single protocol (*i.e.*, UDP) for network probing, *deTector* may miss detecting failures that affect other protocols only, *e.g.*, misconfigured TCP parameters [21]. Adopting other source routing solutions and adding more protocols to increase packet entropy is the future work.

**Loss diagnosis.** While *deTector* can localize where packet drops occur, it does not know what causes the drops, *e.g.*, hardware issues, software bugs, misconfigured rules or bursty traffic. This is a common deficiency in existing monitoring systems, since network diagnosis is rather complex. However, it is possible to automatically distinguish full loss, deterministic partial loss, random partial loss and congestion loss to narrow down diagnosis scope (*e.g.*, using machine learning) because they exhibit different loss characteristics. We consider this as a promising future direction.

## 8 Related Work

**Probe design.** Many existing work (*e.g.*, [9, 13, 42, 29, 22]) exploit events and logs in the control plane (*i.e.*, on switches), or utilize multicasting or network coding, to design probing. Instead, we treat each switch as a blackbox, and adopt a topology-aware end-to-end active probing approach. Some studies [8, 41, 37] aim to minimize the number of probe agents in general Internet setting, while we care more about the number of probe paths and provide some flexibility for placing probe agents. Some existing work [11, 38] aim to select paths to infer the loss rates of all links or all paths, while we aim at identifying only bad links (*i.e.*, failure spots). Estimating loss rates of all links is an indirect and complicated way to do failure localization since it is harder to compute all probe paths. Instead, we design fast algorithms dedicated for localizing faults only. Zeng *et al.* [44] propose a path selection algorithm MSC-$\alpha$ that considers only link coverage and does not scale well if applied to a large DCN. Nicolas *et al.* [18] provide a loss localization solution for backbone networks that does not scale to DCNs, and we do not know their algorithm details either. As compared to the existing work, we further consider evenness in link coverage and identifiability when selecting paths, and propose efficient algorithms that scale well in large data centers.

**Fault localization.** Our goal of identifying faulty links accurately falls squarely under the area of binary network tomography. In binary tomography, it is assumed that network links have one of two states: “good” or “bad”. Approaches based on binary tomography [14, 13, 31] do not apply well to cases that links have partial losses. Fault localization algorithms such as Sherlock [5], Tomo [13], GREEDY [31], SCORE [30] and OMP [40] do not work well for loss localization in data centers, either due to their limited scalability or unsatisfiable assumptions [26]. Herodotou *et al.* [22] use statistical data mining techniques to analyze data collected through active and passive monitoring for fault localization. These work assume uncontrolled measurements, while we combine together two separate steps, path selection and failure localization, to provide accuracy guarantee (§6.4).

**DCN monitoring.** Our work mainly differs from existing monitoring systems such as Pingmesh [21] and NetNORAD [34] in the design of probe matrix. We argue that loss detection and localization must be coupled together to localize more failures (*e.g.*, transient failures) in real time with low overhead. Carefully designed probe matrix is the key to achieve them. LossRadar [36] is a switch-based solution but it requires non-trivial modifications to switch hardware. Combined with network debugging tools such as Everflow [45], we can further root-cause the failures.

## 9 Conclusion

*deTector* is a real-time, low-overhead and high-accuracy monitoring system for large-scale data center networks. At its core is a carefully designed probe matrix, constructed by a greedy path selection algorithm that is scalable and probe overhead minimizing. We also design an efficient failure localization algorithm according to different patterns of packet losses. Our analysis, testbed experiments and large-scale simulations show that *deTector* is highly scalable, practically deployable with low overhead, and can localize failures with high accuracy in near real time.

# Appendix

## A The proof of Theorem 1

We define a function $f(S) = \sum_{path \in S} score[path]$ where $S$ is a path set.

First we prove that $score[path]$ is nonnegative. According to the definition, we need to prove $\sum_{link \in path} w[link] \geq$ # of link sets on *path*. This establishes since the number of links sets on paths can not be more than the number of links of the path.

Since $f(s) \geq 0$, $0 \leq f(S) \leq f(S \cup s)$, i.e., $f(S)$ is nonnegative, monotonically non-decreasing. Since our objective function is to minimize $|S|$, we can achieve it by minimizing $f(S)$.

According to the definition of submodularity, we need to prove $f(s|S) = f(S \cup s) - f(S) \leq f(T \cup s) - f(T) = f(s|T)$ where $S$ and $T$ are path sets and $T \subseteq S$. Note that for the same path, the sum of its link weights is fixed, i.e., equal to path length. $f(s|S) \leq f(s|T)$ establishes because the more paths we have selected, the more link sets we have, and the path is likely to be present in more link sets.

After proving the nonnegativity, monotonicity and submodularity of our objective function, we can conclude the approximation ratio of PMC is $1-\frac{1}{e}$ according to [32].

## B The proof of Theorem 2

**Theorem 2** *The number of paths required to achieve 1-coverage and 1-identifiability in a $k$-ary Fattree is at least* $k^3/5$.

First note that any 1-identifiability probe matrix must cover each link at least once; thus it is also 1-coverage.

Fix any 1-identifiability probe matrix. Let $d(e)$ be the number of paths containing link $e$. Let $\alpha = |\{e : d(e) = 1\}|$ be the number of links that are contained in only one path. Let $n = k^3/2$ be the total number of links in a $k$-ary Fattree. As every link must be contained in at least one path, there are $n-\alpha$ links with $d(e) \geq 2$.

Note that for a link $e$ with $d(e) = 1$, we must have $d(e') \geq 2$ if $e'$ is contained in the same path as $e$, otherwise they are not distinguishable.

As every path contains exactly 4 links, the required number of paths is at least

$$\frac{1}{4}\sum_e d(e) \geq \frac{\alpha + \max\{3\alpha, 2(n-\alpha)\}}{4} \geq \frac{2}{5}n,$$

where equality is achieved when $\alpha = \frac{2}{5}n = k^3/5$.

## References

[1] Fbtracert. https://github.com/facebook/fbtracert, 2016.

[2] Microsoft Netbouncer. https://www.youtube.com/watch?v=nfEOEKlInK8, 2016.

[3] The deTector project. https://github.com/atc2017deTector/deTector, 2017.

[4] Al-Fares, M., Loukissas, A., and Vahdat, A. A scalable, commodity data center network architecture. In *Proc. of ACM SIGCOMM* (2008).

[5] 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 *Proc. of ACM SIGCOMM* (2007).

[6] Benson, T. Data set for IMC 2010 data center measurement. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html, 2010.

[7] Benson, T., Anand, A., Akella, A., and Zhang, M. Understanding data center traffic characteristics. In *Proc. of ACM SIGCOMM* (2010).

[8] Brodie, M., Rish, I., and Ma, S. Optimizing probe selection for fault localization. In *Proc. of the 12th International Workshop on Distributed Systems: Operations and Management (DSOM)* (2001).

[9] Castro, R., Coates, M., Liang, G., Nowak, R., and Yu, B. Network tomography: recent developments. *Statistical Science 19*, 3 (2004), 499–517.

[10] Chen, K., Guo, C., Wu, H., Yuan, J., Feng, Z., Chen, Y., Lu, S., and Wu, W. Generic and automatic address configuration for data center networks. In *Proc. of ACM SIGCOMM* (2010).

[11] Chen, Y., Bindel, D., Song, H., and Katz, R. H. An algebraic approach to practical and scalable overlay network monitoring. In *Proc. of ACM SIGCOMM* (2004).

[12] Darga, P. T., Sakallah, K. A., and Markov, I. L. Faster symmetry discovery using sparsity of symmetries. In *Proc. of the 45th Annual Design Automation Conference (DAC)* (2008).

[13] Dhamdhere, A., Teixeira, R., Dovrolis, C., and Diot, C. Netdiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *Proc. of the 3rd ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2007).

[14] Duffield, N. Network tomography of binary network performance characteristics. *IEEE Transactions on Information Theory 52*, 12 (November 2006), 5373–5388.

[15] Gill, P., Jain, N., and Nagappan, N. Understanding network failures in data centers: measurement, analysis, and implications. In *Proc. of ACM SIGCOMM* (2011).

[16] Govindan, R., Minei, I., Kallahalla, M., Koley, B., and Vahdat, A. Evolve or die: High-availability design principles drawn from Google's network infrastructure. In *Proc. of ACM SIGCOMM* (2016).

[17] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. VL2: a scalable and flexible data center network. In *Proc. of ACM SIGCOMM* (2009).

[18] Guilbaud, N., and Cartlidge, R. Localizing packet loss in a large complex network. https://www.nanog.org/meetings/nanog57/presentations/Tuesday/tues.general.GuilbaudCartlidge.Topology.7.pdf, 2013.

[19] Guo, C., Lu, G., Li, D., Wu, H., Zhang, X., Shi, Y., Tian, C., Zhang, Y., and Lu, S. BCube: a high performance, server-centric network architecture for modular data centers. In *Proc. of ACM SIGCOMM* (2009).

[20] Guo, C., Wu, H., Tan, K., Shi, L., Zhang, Y., and Lu, S. DCell: a scalable and fault-tolerant network structure for data centers. In *Proc. of ACM SIGCOMM* (2008).

[21] 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 *Proc. of ACM SIGCOMM*, 2015.

[22] Herodotou, H., Ding, B., Balakrishnan, S., Outhred, G., and Fitter, P. Scalable near real-time failure localization of data center networks. In *Proc. of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2014).

[23] Hu, C., Yang, J., Gong, Z., Deng, S., and Zhao, H. DesktopDC: setting all programmable data center networking testbed on desk. *ACM SIGCOMM Computer Communication Review 44*, 4 (2015), 593–594.

[24] Hu, C., Yang, J., Zhao, H., and Lu, J. Design of all programmable innovation platform for software defined networking. In *Proc. of Open Networking Summit*, 2014.

[25] 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. In *Proc. of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2015).

[26] Huang, Y., Feamster, N., and Teixeira, R. Practical issues with using network tomography for fault diagnosis. *ACM SIGCOMM Computer Communication Review 38*, 5 (October 2008), 53–58.

[27] Isard, M. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review - Systems work at Microsoft Research 41*, 2 (April 2007), 60–67.

[28] Jyothi, S. A., Dong, M., and Godfrey, P. Towards a flexible data center fabric with source routing. In *Proc. of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)* (2015).

[29] Kandula, S., Mahajan, R., Verkaik, P., Agarwal, S., Padhye, J., and Bahl, P. Detailed diagnosis in enterprise networks. In *Proc. of ACM SIGCOMM* (2009).

[30] Kompella, R. R., Yates, J., Greenberg, A., and Snoeren, A. C. IP fault localization via risk modeling. In *Proc. of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2005).

[31] Kompella, R. R., Yates, J., Greenberg, A., and Snoeren, A. C. Detection and localization of network black holes. In *Proc. of IEEE INFOCOM* (2007).

[32] Kun, J. Submodularity and approximation. goo.gl/9GFFTW, July 2014.

[33] Lapukhov, P. Configuring IPSLA. http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst4500/12-2/44sg/configuration/guide/Wrapper-44SG/swipsla.html.

[34] Lapukhov, P. Network debugging at scale. https://www.nanog.org/sites/default/files/Lapukhov_Move_Fast_Unbreak.pdf, 2016.

[35] Leskovec, J., Krause, A., Guestrin, C., Faloutsos, C., VanBriesen, J., and Glance, N. Cost-effective outbreak detection in networks. In *Proc. of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2007).

[36] Liú, Y., Miao, R., Kim, C., and Yuú, M. LossRadar: Fast detection of lost packets in data center networks. In *Proc. of the 12th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2016).

[37] Ma, L., He, T., Leung, K. K., Swami, A., and Towsley, D. Monitor placement for maximal identifiability in network tomography. In *Proc. of IEEE INFOCOM* (2014).

[38] Ma, L., He, T., Leung, K. K., Towsley, D., and Swami, A. Efficient identification of additive link metrics via network tomography. In *Proc. of the 33rd IEEE International Conference on Distributed Computing Systems (ICDCS)* (2013).

[39] Patel, P., Bansal, D., Yuan, L., Murthy, A., Greenberg, A., Maltz, D. A., Kern, R., Kumar, H., Zikos, M., Wu, H., et al. Ananta: cloud scale load balancing. In *Proc. of ACM SIGCOMM* (2013).

[40] Pati, Y. C., Rezaiifar, R., and Krishnaprasad, P. Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In *Proc. of the 27th Asilomar Conference on Signals, Systems and Computers* (1993).

[41] Sadaphal, V., Natu, M., Vin, H., and Shenoy, P. Varanus: More-with-less fault localization in data centers. In *Proc. of the 4th International Conference on Communication Systems and Networks* (2012).

[42] Sharma, G., Jaggi, S., and Dey, B. Network tomography via network coding. In *Proc. of the 2008 Information Theory and Applications Workshop (ITA)* (2008).

[43] 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 *Proc. of ACM SIGCOMM* (2015).

[44] Zeng, H., Mahajan, R., McKeown, N., Varghese, G., Yuan, L., and Zhang, M. Measuring and troubleshooting large operational multipath networks with gray box testing, msr-tr-2015-55. Tech. rep., 2015.

[45] Zhu, Y., Kang, N., Cao, J., Greenberg, A., Lu, G., Mahajan, R., Maltz, D., Yuan, L., Zhang, M., Zhao, B. Y., et al. Packet-level telemetry in large datacenter networks. In *Proc. of ACM SIGCOMM* (2015).