007: Democratically Finding the Cause of Packet Drops

Behnaz Arzani, *Microsoft Research;* **Selim Ciraci,** *Microsoft;* **Luiz Chamon,** *University of Pennsylvania;* **Yibo Zhu and Hongqiang (Harry) Liu,** *Microsoft Research;* **Jitu Padhye,** *Microsoft;* **Boon Thau Loo,** *University of Pennsylvania;* **Geoff Outhred,** *Microsoft*

https://www.usenix.org/conference/nsdi18/presentation/arzani

This paper is included in the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18).

April 9–11, 2018 • Renton, WA, USA

ISBN 978-1-931971-43-0

Open access to the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation is sponsored by USENIX.

007: Democratically Finding The Cause of Packet Drops

Behnaz Arzani[1], Selim Ciraci[2], Luiz Chamon[3], Yibo Zhu[1], Hongqiang (Harry) Liu[1], Jitu Padhye[2], Boon Thau Loo[3], Geoff Outhred[2]
[1]*Microsoft Research,*[2]*Microsoft,*[3]*University of Pennsylvania*

Abstract – Network failures continue to plague datacenter operators as their symptoms may not have direct correlation with where or why they occur. We introduce 007, a lightweight, always-on diagnosis application that can find problematic links and also pinpoint problems *for each TCP connection*. 007 is completely contained within the end host. During its two month deployment in a tier-1 datacenter, it detected every problem found by previously deployed monitoring tools while also finding the sources of other problems previously undetected.

1 Introduction

007 has an ambitious goal: for every packet drop on a TCP flow in a datacenter, find the link that dropped the packet and do so with negligible overhead and no changes to the network infrastructure.

This goal may sound like an overkill—after all, TCP is supposed to be able to deal with a few packet losses. Moreover, packet losses might occur due to congestion instead of network equipment failures. Even network failures might be transient. Above all, there is a danger of drowning in a sea of data without generating any actionable intelligence.

These objections are valid, but so is the need to diagnose "failures" that can result in severe problems for applications. For example, in our datacenters, VM images are stored in a storage service. When a VM boots, the image is mounted over the network. Even a small network outage or a few lossy links can cause the VM to "panic" and reboot. In fact, 17% of our VM reboots are due to network issues and in over 70% of these none of our monitoring tools were able to find the links that caused the problem.

VM reboots affect customers and we need to understand their root cause. Any persistent pattern in such transient failures is a cause for concern and is potentially actionable. One example is silent packet drops [1]. These types of problems are nearly impossible to detect with traditional monitoring tools (e.g., SNMP). If a switch is experiencing these problems, we may want to reboot or replace it. These interventions are "costly" as they affect a large number of flows/VMs. Therefore, careful blame assignment is necessary. Naturally, this is only one example that would benefit from such a detection system.

There is a lot of prior work on network failure diagnosis, though one of the existing systems meet our ambitious goal. Pingmesh [1] sends periodic probes to detect failures and can leave "gaps" in coverage, as it must manage the overhead of probing. Also, since it uses out-of-band probes, it cannot detect failures that affect only in-band data. Roy et al. [2] monitor all paths to detect failures but require modifications to routers and special features in the switch (§10). Everflow [3] can be used to find the location of packet drops but it would require capturing all traffic and is not scalable. We asked our operators what would be the most useful solution for them. Responses included: "In a network of $\geq 10^6$ links its a reasonable assumption that there is a non-zero chance that a number (> 10) of these links are bad (due to device, port, or cable, etc.) and we cannot fix them simultaneously. Therefore, fixes need to be prioritized based on customer impact. However, currently we do not have a direct way to correlate customer impact with bad links". This shows that current systems do not satisfy operator needs as they do not provide application and connection level context.

To address these limitations, we propose 007, a simple, lightweight, always-on monitoring tool. 007 records the path of TCP connections (flows) suffering from one or more retransmissions and assigns proportional "blame" to each link on the path. It then provides a *ranking* of links that represents their relative drop rates. Using this ranking, it can find the most likely cause of drops in each TCP flow.

007 has several noteworthy properties. First, it does not require any changes to the existing networking infrastructure. Second, it does not require changes to the client software—the monitoring agent is an independent entity that sits on the side. Third, it detects in-band failures. Fourth, it continues to perform well in the presence of noise (e.g. lone packet drops). Finally, it's overhead is negligible.

While the high-level design of 007 appear simple, the practical challenges of making 007 work and the theoretical challenge of *proving* it works are non-trivial. For example, its path discovery is based on a traceroute-like approach. Due to the use of ECMP, traceroute packets have to be carefully crafted to ensure that they follow the same path as the TCP flow. Also, we must ensure that we do not overwhelm routers by sending too many traceroutes (traceroute responses are handled by control-plane CPUs of routers, which are quite puny). Thus, we

need to ensure that our sampling strikes the right balance between accuracy and the overhead on the switches. On the theoretical side, we are able to show that 007's simple blame assignment scheme is highly accurate even in the presence of noise.

We make the following contributions: (i) we design 007, a simple, lightweight, and yet accurate fault localization system for datacenter networks; (ii) we prove that 007 is accurate without imposing excessive burden on the switches; (iii) we prove that its blame assignment scheme correctly finds the failed links with high probability; and (iv) we show how to tackle numerous practical challenges involved in deploying 007 in a real datacenter.

Our results from a two month deployment of 007 in a datacenter show that it finds all problems found by other previously deployed monitoring tools while also finding the sources of problems for which information is not provided by these monitoring tools.

2 Motivation

007 aims to identify the cause of retransmissions with high probability. It is is driven by two practical requirements: (i) it should scale to datacenter size networks and (ii) it should be deployable in a running datacenter with as little change to the infrastructure as possible. Our current focus is mainly on analyzing infrastructure traffic, especially connections to services such as storage as these can have severe consequences (see §1, [4]). Nevertheless, the same mechanisms can be used in other contexts as well (see §9). We deliberately include congestion-induced retransmissions. If episodes of congestion, however short-lived, are common on a link, we want to be able to flag them. Of course, in practice, any such system needs to deal with a certain amount of noise, a concept we formalize later.

There are a number of ways to find the cause of packet drops. One can monitor switch counters. These are inherently unreliable [5] and monitoring thousands of switches at a fine time granularity is not scalable. One can use new hardware capabilities to gather more useful information [6]. Correlating this data with each retransmission *reliably* is difficult. Furthermore, time is needed until such hardware is production-ready and switches are upgraded. Complicating matters, operators may be unwilling to incur the expense and overhead of such changes [4]. One can use PingMesh [1] to send probe packets and monitor link status. Such systems suffer from a rate of probing trade-off: sending too many probes creates unacceptable overhead whereas reducing the probing rate leaves temporal and spatial gaps in coverage. More importantly, the probe traffic does not capture what the end user and TCP flows see. Instead, we choose to use data traffic itself as probe traffic. Using data traffic has the advantage that the system introduces little to no monitoring overhead.

As one might expect, almost all traffic in our datacenters is TCP traffic. One way to monitor TCP traffic is to use a system like Everflow. Everflow inserts a special tag in every packet and has the switches mirror tagged packets to special collection servers. Thus, if a tagged packet is dropped, we can determine the link on which it happened. Unfortunately, there is no way to know in advance which packet is going to be dropped, so we would have to tag and mirror every TCP packet. This is clearly infeasible. We could tag only a fraction of packets, but doing so would result in another sampling rate trade-off. Hence, we choose to rely on some form of network tomography [7, 8, 9]. We can take advantage of the fact that TCP is a connection-oriented, reliable delivery protocol so that any packet loss results in retransmissions that are easy to detect.

If we knew the path of all flows, we could set up an optimization to find which link dropped the packet. Such an optimization would minimize the number of "blamed" links while simultaneously explaining the cause of all drops. Indeed past approaches such as MAX COVERAGE and Tomo [10, 11] aim to approximate the solution of such an optimization (see §12 for an example). There are problems with this approach: (i) the optimization is NP-hard [12]. Solving it on a datacenter scale is infeasible. (ii) tracking the path of every flow in the datacenter is not scalable in our setting. We can use alternative solutions such as Everflow or the approach of [2] to track the path of SYN packets. However, both rely on making changes to the switches. The only way to find the path of a flow without any special infrastructure support is to employ something like a traceroute. Traceroute relies on getting ICMP TTL exceeded messages back from the switches. These messages are generated by the control-plane, i.e., the switch CPU. To avoid overloading the CPU, our administrators have capped the rate of ICMP responses to 100 per second. This severely limits the number of flows we can track.

Given these limitations, what can we do? We analyzed the drop patterns in two of our datacenters and found: typically when there are packet drops, multiple flows experience drops. We show this in Figure 1a for TCP flows in production datacenters. The figure shows the number of flows experiencing drops in the datacenter conditioned on the total number of packets dropped in that datacenter in 30 second intervals. The data spans one day. We see that the more packets are dropped in the datacenter,

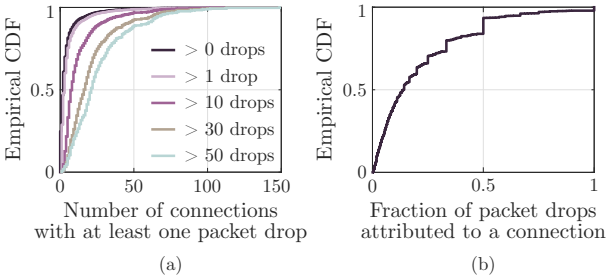


Figure 1: Observations from a production network: (a) CDF of the number of flows with at least one retransmission; (b) CDF of the fraction of drops belonging to each flow in each 30 second interval.

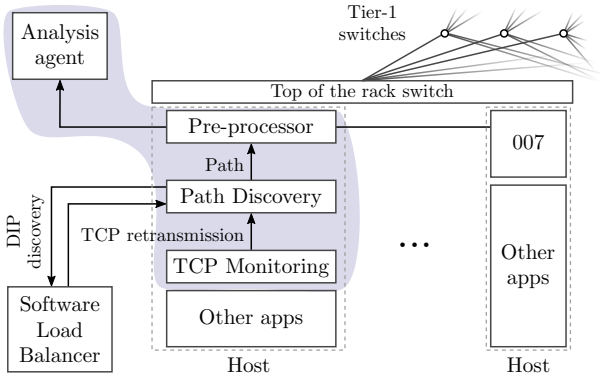


Figure 2: Overview of 007 architecture

the more flows experience drops and 95% of the time, at least 3 flows see drops when we condition on ≥ 10 total drops. We focus on the ≥ 10 case because lower values mostly capture noisy drops due to one-off packet drops by healthy links. In most cases drops are distributed across flows and no single flow sees more than 40% of the total packet drops. This is shown in Figure 1b (we have discarded all flows with 0 drops and cases where the total number of drops was less than 10). We see that in $\geq 80\%$ of cases, no single flow captures more than 34% of all drops.

Based on these observations and the high path diversity in datacenter networks [13], we show that if: (a) we only track the path of those flows that have retransmissions, (b) assign each link on the path of such a flow a vote of $1/h$, where h is the path length, and (c) sum up the votes during a given period, then the top-voted links are almost always the ones dropping packets (see §5)! Unlike the optimization, our scheme is able to provide a *ranking* of the links in terms of their drop rates, i.e. if link A has a higher vote than B, it is also dropping more packets (with high probability). This gives us a heat-map of our network which highlights the links with the most impact to a *given application/customer* (because we know which links impact a particular flows).

3 Design Overview

Figure 2 shows the overall architecture of 007. It is deployed alongside other applications on each end-host as a user-level process running in the host OS. 007 consists of three agents responsible for TCP monitoring, path discovery, and analysis.

The *TCP monitoring agent* detects retransmissions at each end-host. The Event Tracing For Windows (ETW) [14] framework[1] notifies the agent as soon as an active flow suffers a retransmission.

Upon a retransmission, the monitoring agent triggers the *path discovery agent* (§4) which identifies the flow's path to the destination IP (DIP).

At the end-hosts, a voting scheme (§5) is used based on the paths of flows that had retransmissions. At regular intervals of 30s the votes are tallied by a centralized *analysis agent* to find the top-voted links. Although we use an aggregation interval of 30s, failures *do not* have to last for 30s.

007's implementation consists of 6000 lines of C++ code. Its memory usage never goes beyond 600 KB on any of our production hosts, its CPU utilization is minimal (1-3%), and its bandwidth utilization due to traceroute is minimal (maximum of 200 KBps per host). 007 is proven to be accurate (§5) in typical datacenter conditions (a full description of the assumed conditions can be found in §9).

4 The Path Discovery Agent

The path discovery agent uses traceroute packets to find the path of flows that suffer retransmissions. These packets are used solely to identify the path of a flow. They do not need to be dropped for 007 to operate. We first ensure that the number of traceroutes sent by the agent does not overload our switches (§4.1). Then, we briefly describe the key engineering issues and how we solve them (§4.2).

4.1 ICMP Rate Limiting

Generating ICMP packets in response to traceroute consumes switch CPU, which is a valuable resource. In our network, there is a cap of $T_{\max} = 100$ on the number of ICMP messages a switch can send per second. To ensure that the traceroute load does not exceed $T_{\max}$, we start by noticing that a small fraction of flows go through tier-3 switches (T_3). Indeed, after monitoring all TCP flows in our network for one hour, only 2.1% went through a T_3 switch. Thus we can ignore T_3 switches in our analysis. Given that our network is a Clos topology and assuming that hosts under a top of the rack switch (ToR) communicate with hosts under a different ToR uniformly at random (see §6 for when this is not the case):

[1] Similar functionality exists in Linux.

Theorem 1. *The rate of ICMP packets sent by any switch due to a traceroute is below $T_{\max}$ if the rate C_t at which hosts send traceroutes is upper bounded as*

$$C_t \leq \frac{T_{\max}}{n_0 H} \min\left[n_1, \frac{n_2(n_0 n_{\text{pod}} - 1)}{n_0(n_{\text{pod}} - 1)}\right], \quad (1)$$

where n_0, n_1, and n_2, are the numbers of ToR, T_1, and T_2 switches respectively, n_{pod} is the number of pods, and H is the number of hosts under each ToR.

See §12 for proof. The upper bound of C_t in our datacenters is 10. As long as hosts do not have more than 10 flows *with retransmissions* per second, we can *guarantee* that the number of traceroutes sent by 007 will not go above $T_{\max}$. We use C_t as a threshold to limit the traceroute rate of each host. Note that there are two independent rate limits, one set at the host by 007 and the other set by the network operators on the switch ($T_{\max}$). Additionally, the agent triggers path discovery *for a given connection* no more than once every epoch to further limit the number of traceroutes. We will show in §5 that this number is sufficient to ensure high accuracy.

4.2 Engineering Challenges

Using the correct five-tuple. As in most datacenters, our network also uses ECMP. All packets of a given flow, defined by the five-tuple, follow the same path [15]. Thus, traceroute packets must have the same five-tuple as the flow we want to trace. To ensure this, we must account for load balancers.

TCP connections are initiated in our datacenter in a way similar to that described in [16]. The connection is first established to a virtual IP (VIP) and the SYN packet (containing the VIP as destination) goes to a software load balancer (SLB) which assigns that flow to a physical destination IP (DIP) and a service port associated with that VIP. The SLB then sends a configuration message to the virtual switch (vSwitch) in the hypervisor of the source machine that registers that DIP with that vSwitch. The destination of all subsequent packets in that flow have the DIP as their destination and do not go through the SLB. For the path of the traceroute packets to match that of the data packets, its header should contain the DIP and not the VIP. Thus, before tracing the path of a flow, the path discovery agent first queries the SLB for the VIP-to-DIP mapping for that flow. An alternative is to query the vSwitch. In the instances where the failure also results in connection termination the mapping may be removed from the vSwitch table. It is therefore more reliable to query the SLB. Note that there are cases where the TCP connection establishment itself may fail due to packet loss. Path discovery is not triggered for such connections. It is also not triggered when the query to the SLB fails to avoid tracerouting the internet.

Re-routing and packet drops. Traceroute itself may fail. This may happen if the link drop rate is high or due to a blackhole. This actually helps us, as it directly pinpoints the faulty link and our analysis engine (§5) is able to use such partial traceroutes.

A more insidious possibility is that routing may change by the time traceroute starts. We use BGP in our datacenter and a lossy link may cause one or more BGP sessions to fail, triggering rerouting. Then, the traceroute packets may take a different path than the original connection. However, RTTs in a datacenter are typically less than 1 or 2 ms, so TCP retransmits a dropped packet quickly. The ETW framework notifies the monitoring agent immediately, which invokes the path discovery agent. The only additional delay is the time required to query the SLB to obtain the VIP-to-DIP mapping, which is typically less than a millisecond. Thus, as long as paths are stable for a few milliseconds after a packet drop, the traceroute packets will follow the same path as the flow and the probability of error is low. Past work has shown this to be usually the case [17].

Our network also makes use of link aggregation (LAG) [18]. However, unless all the links in the aggregation group fail, the L3 path is not affected.

Router aliasing [19]. This problem is easily solved in a datacenter, as we know the topology, names, and IPs of all routers and interfaces. We can simply map the IPs from the traceroutes to the switch names.

To summarize, 007's path discovery implementation is as follows: Once the TCP monitoring agent notifies the path discovery agent that a flow has suffered a retransmission, the path discovery agent checks its cache of discovered path for that epoch and if need be, queries the SLB for the DIP. It then sends 15 appropriately crafted TCP packets with TTL values ranging from 0–15. In order to disambiguate the responses, the TTL value is also encoded in the IP ID field [20]. This allows for concurrent traceroutes to multiple destinations. The TCP packets deliberately carry a bad checksum so that they do not interfere with the ongoing connection.

5 The Analysis Agent

Here, we describe 007's analysis agent focusing on its voting-based scheme. We also present alternative NP-hard optimization solutions for comparison.

5.1 Voting-Based Scheme

007's analysis agent uses a simple voting scheme. If a flow sees a retransmission, 007 votes its links as *bad*. Each vote has a value that is tallied at the end of every epoch, providing a natural ranking of the links. We set the value of good votes to 0 (if a flow has no

retransmission, no traceroute is needed). Bad votes are assigned a value of $\frac{1}{h}$, where h is the number of hops on the path, since each link on the path is equally likely to be responsible for the drop.

The ranking obtained after compiling the votes allows us to identify the most likely cause of drops on each flow: links ranked higher have higher drop rates (Theorem 2). To further guard against high levels of noise, we can use our knowledge of the topology to adjust the links votes. Namely, we iteratively pick the most voted link $l_{\max}$ and estimate the portion of votes obtained by all other links due to failures on $l_{\max}$. This estimate is obtained for each link k by (i) assuming all flows having retransmissions and going through $l_{\max}$ had drops due to $l_{\max}$ and (ii) finding what fraction of these flows go through k by assuming ECMP distributes flows uniformly at random. Our evaluations showed that this results in a 5% reduction in false positives.

Algorithm 1 Finding the most problematic links in the network.

```
1:  L ← Set of all links
2:  P ← Set of all possible paths
3:  v(l_i) ← Number of votes for l_i ∈ L
4:  B ← Set of most problematic links
5:  l_max ← Link with maximum votes in ∀l_i ∈ L ∩ B^c
6:  while v(l_max) ≥ 0.01(Σ_{l_i ∈ L} v(l_i)) do
7:      l_max ← argmax_{l_i ∈ L ∩ B^c} v(l_i)
8:      B ← B ∪ {l_max}
9:      for l_i ∈ L ∩ B^c do
10:         if ∃ p_i ∈ P s.t. l_i ∈ p_i & l_max ∈ p_i then
11:             Adjust the score of l_i
12:         end if
13:     end for
14: end while
15: return B
```

007 can also be used to detect failed links using Algorithm 1. The algorithm sorts the links based on their votes and uses a threshold to determine if there are problematic links. If so, it adjusts the votes of all other links and repeats until no link has votes above the threshold. In Algorithm 1, we use a threshold of 1% of the total votes cast based on a parameter sweep where we found that it provides a reasonable trade-off between precision and recall. Higher values reduce false positives but increase false negatives.

Here we have focused on detecting link failures. 007 can also be used to detect switch failures in a similar fashion by applying votes to switches instead of links. This is beyond the scope of this work.

5.2 Voting Scheme Analysis

Can 007 deliver on its promise of finding the most probable cause of packet drops on each flow? This is not trivial. In its voting scheme, failed connections contribute to increase the tally of both good and bad links. Moreover, in a large datacenter such as ours, occasional, lone, and sporadic drops can and *will* happen due to good links. These failures are akin to noise and can cause severe inaccuracies in any detection system [21], 007 included. We show that the likelihood of 007 making these errors is small. Given our topology (Clos):

Theorem 2. *For $n_{\text{pod}} \geq \frac{n_0}{n_1} + 1$, 007 will find with probability $1 - 2e^{-\mathcal{O}(N)}$ the $k < \frac{n_2(n_0 n_{\text{pod}} - 1)}{n_0(n_{\text{pod}} - 1)}$ bad links that drop packets with probability p_b among good links that drop packets with probability p_g if*

$$p_g \leq (n_u \alpha)^{-1} \left[1 - (1 - p_b)^{n_l}\right],$$

where N is the total number of flows between hosts, n_l and n_u are lower and upper bounds, respectively, on the number of packets per connection, and

$$\alpha = \frac{n_0(4n_0 - k)(n_{\text{pod}} - 1)}{n_2(n_0 n_{\text{pod}} - 1) - n_0(n_{\text{pod}} - 1)k}. \qquad (2)$$

The proof is deferred to the appendices due to space constraints. Theorem 2 states that under mild conditions, links with higher drop rates are ranked higher by 007. Since a single flow is unlikely to go through more than one failed link in a network with thousands of links, it allows 007 to find the most likely cause of packet drops on each flow.

A corollary of Theorem 2 is that in the absence of noise ($p_g = 0$), 007 can find all bad links with high probability. In the presence of noise, 007 can still identify the bad links as long as the probability of dropping packets on non-failed links is low enough (the signal-to-noise ratio is large enough). This number is compatible with typical values found in practice. As an example, let n_l and n_u be the 10^{th} and 90^{th} percentiles respectively of the number of packets sent by TCP flows across all hosts in a 3 hour period. If $p_b \geq 0.05\%$, the drop rate on good links can be as high as 1.8×10^{-6}. Drop rates in a production datacenter are typically below 10^{-8} [22].

Another important consequence of Theorem 2 is that it establishes that the probability of errors in 007's results diminishes exponentially with N, so that even with the limits imposed by Theorem 1 we can accurately identify the failed links. The conditions in Theorem 2 are sufficient but not necessary. In fact, §6 shows how well 007 performs even when the conditions in Theorem 2 do not hold.

5.3 Optimization-Based Solutions

One of the advantages of 007's voting scheme is its simplicity. Given additional time and resources we may consider searching for the optimal sets of failed links by finding the most likely cause of drops given the available evidence. For instance, we can find the

least number of links that explain all failures as we know the flows that had packet drops and their path. This can be written as an optimization problem we call the *binary program*. Explicitly,

$$\begin{aligned} \text{minimize} \quad & \|\boldsymbol{p}\|_0 \\ \text{subject to} \quad & \boldsymbol{A}\boldsymbol{p} \geq \boldsymbol{s} \\ & \boldsymbol{p} \in \{0,1\}^L \end{aligned} \tag{3}$$

where $\boldsymbol{A}$ is a $C \times L$ routing matrix; $\boldsymbol{s}$ is a $C \times 1$ vector that collects the status of each flow during an epoch (each element of $\boldsymbol{s}$ is 1 if the connection experienced at least one retransmission and 0 otherwise); L is the number of links; C is the number of connections in an epoch; and $\|\boldsymbol{p}\|_0$ denotes the number of nonzero entries of the vector $\boldsymbol{p}$. Indeed, if the solution of (3) is $\boldsymbol{p}^\star$, then the i-th element of $\boldsymbol{p}^\star$ indicates whether the binary program estimates that link i failed.

Problem (3) is the NP-hard minimum set covering problem [23] and is intractable. Its solutions can be approximated greedily as in MAX COVERAGE or Tomo [10, 11] (see appendix). For benchmarking, we compare 007 to the true solution of (3) obtained by a mixed-integer linear program (MILP) solver [24]. Our evaluations showed that 007 (Algorithm 1) significantly outperforms this binary optimization (by more than 50% in the presence of noise). We illustrate this point in Figures 4 and 10, but otherwise omit results for this optimization in §6 for clarity.

The binary program (3) does not provide a ranking of links. We also consider a solution in which we determine the number of packets dropped by each link, thus creating a natural ranking. The *integer program* can be written as

$$\begin{aligned} \text{minimize} \quad & \|\boldsymbol{p}\|_0 \\ \text{subject to} \quad & \boldsymbol{A}\boldsymbol{p} \geq \boldsymbol{c} \\ & \|\boldsymbol{p}\|_1 = \|\boldsymbol{c}\|_1 \\ & p_i \in \mathbb{N} \cup \{0\} \end{aligned} \tag{4}$$

where $\mathbb{N}$ is the set of natural numbers and $\boldsymbol{c}$ is a $C \times 1$ vector that collects the number of retransmissions suffered by each flow during an epoch. The solution $\boldsymbol{p}^\star$ of (4) represents the number of packets dropped by each link, which provides a ranking. The constraint $\|\boldsymbol{p}\|_1 = \|\boldsymbol{c}\|_1$ ensures each failure is explained only once. As with (3), this problem is NP-hard [12] and is only used as a benchmark. As it uses more information than the binary program (the number of failures), (4) performs better (see §6).

In the next three sections, we present our evaluation of 007 in simulations (§6), in a test cluster (§7), and in one of our production datacenters (§8).

6 Evaluations: Simulations

We start by evaluating in simulations where we know the ground truth. 007 first finds flows whose drops were due to noise and marks them as "noise drops". It then finds the link most likely responsible for drops on the remaining set of flows ("failure drops"). A noisy drop is defined as one where the corresponding link only dropped a single packet. 007 never marked a connection into the noisy category incorrectly. We therefore focus on the accuracy for connections that 007 puts into the failure drop class.

Performance metrics. Our measure for the performance of 007 is *accuracy*, which is the proportion of correctly identified drop causes. For evaluating Algorithm 1, we use *recall* and *precision*. Recall is a measure of reliability and shows how many of the failures 007 can detect (false negatives). For example, if there are 100 failed links and 007 detects 90 of them, its recall is 90%. Precision is a measure of accuracy and shows to what extent 007's results can be trusted (false positives). For example, if 007 flags 100 links as bad, but only 90 of those links actually failed, its precision is 90%.

Simulation setup. We use a flow level simulator [25] implemented in MATLAB. Our topology consists of 4160 links, 2 pods, and 20 ToRs per pod. Each host establishes 2 connections per second to a random ToR outside of its rack. The simulator has two types of links. For *good links*, packets are dropped at a very low rate chosen uniformly from $(0, 10^{-6})$ to simulate noise. On the other hand, *failed links* have a higher drop rate to simulate failures. By default, drop rates on failed links are set to vary uniformly from 0.01% to 1%, though to study the impact of drop rates we do allow this rate to vary as an input parameter. The number of good and failed links is also tunable. Every 30 seconds of simulation time, we send up to 100 packets per flow and drop them based on the rates above as they traverse links along the path. The simulator records all flows with at least one drop and for each such flow, the link with the most drops.

We compare 007 against the solutions described in §5.3. We only show results for the binary program (3) in Figures 4 and 10 since its performance is typically inferior to 007 and the integer program (4) due to noise. This also applies to MAX COVERAGE or Tomo [10, 11, 26] *as they are approximations of the binary program* (see [27]).

6.1 In The Optimal Case

The bounds of Theorem 2 are sufficient (not necessary) conditions for accuracy. We first validate that 007 can achieve high levels of accuracy as expected when these bounds hold. We set the drop rates on the failed links to be between $(0.05\%, 1\%)$. We refer the

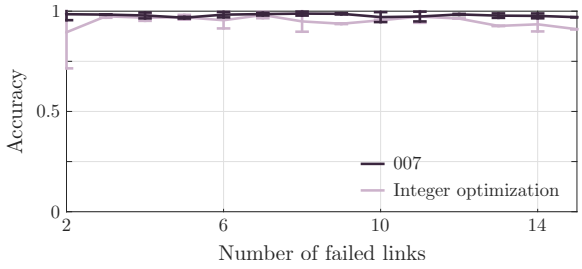


Figure 3: When Theorem 2 holds.

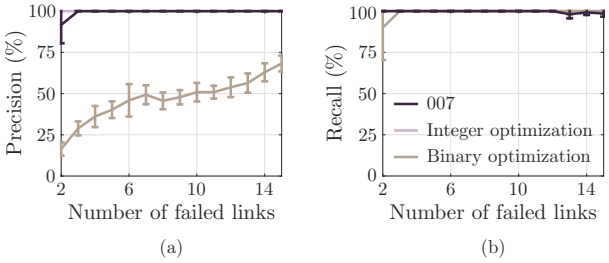


Figure 4: Algorithm 1 when Theorem 2 holds.

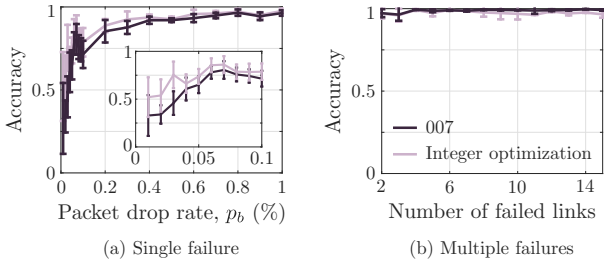


Figure 5: 007's accuracy for varying drop rates.

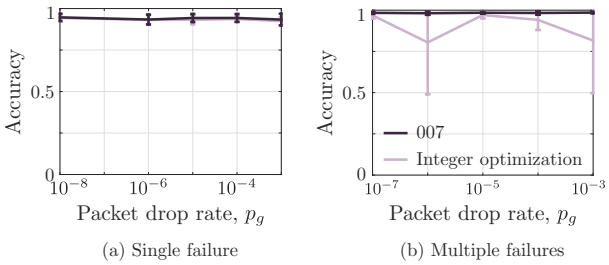


Figure 6: 007's accuracy for varying noise levels.

reader to [2] for why these drop rates are reasonable.

Accuracy. Figure 3 shows that 007 has an average accuracy that is higher than 96% in almost all cases. Due to its robustness to noise, it also outperforms the optimization algorithm (§ 5.3) in most cases.

Recall & precision. Figure 4 shows that even when failed links have low packet drop rates, 007 detects them with high recall/precision.

We proceed to evaluate 007's accuracy when the bounds in Theorem 2 *do not* hold. This shows these conditions are not necessary for good performance.

6.2 Varying Drop Rates

Our next experiment aims to push the boundaries of Theorem 2 by varying the "failed" links drop rates below the conservative bounds of Theorem 2.

Single Failure. Figure 5a shows results for different drop rates on a single failed link. It shows that 007 can find the cause of drops on each flow with high accuracy. Even as the drop rate decreases below the bounds of Theorem 2, we see that 007 can maintain accuracy on par with the optimization.

Multiple Failures. Figure 5b shows that 007 is successful at finding the link responsible for a drop even when links have very different drop rates. Prior work have reported the difficulty of detecting such cases [2]. However, 007's accuracy remains high.

6.3 Impact of Noise

Single Failure. We vary noise levels by changing the drop rate of good links. We see that higher noise levels have little impact on 007's ability to find the cause of drops on individual flows (Figure 6a).

Multiple Failures. We repeat this experiment for the case of 5 failed links. Figure 6b shows the results. 007 shows little sensitivity to the increase in noise when finding the cause of per-flow drops. Note that the large confidence intervals of the optimization is a result of its high sensitivity to noise.

6.4 Varying Number of Connections

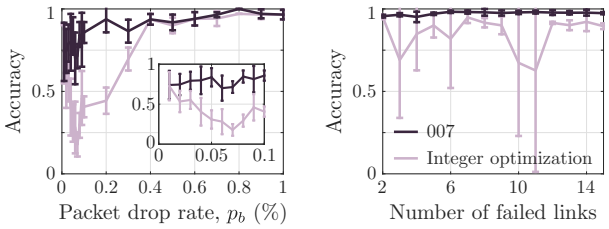
In previous experiments, hosts opened 60 connections per epoch. Here, we allow hosts to choose the number of connections they create per epoch uniformly at random between $(10, 60)$. Recall, from Theorem 2, that a larger number of connections from each host helps 007 improve its accuracy.

Single Failure. Figure 7a shows the results. 007 accurately finds the cause of packet drops on each connection. It also outperforms the optimization when the failed link has a low drop rate. This is because the optimization has multiple optimal points and is not sufficiently constrained.

Multiple Failures. Figure 7b shows the results for multiple failures. The optimization suffers from the lack of information to constrain the set of results. It therefore has a large variance (confidence intervals). 007 on the other hand maintains high probability of detection no matter the number of failures.

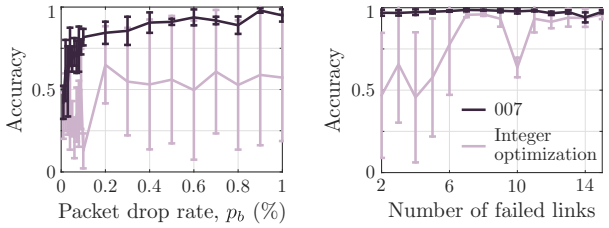
6.5 Impact of Traffic Skews

Single Failure. We next demonstrate 007's ability to detect the cause of drops even under heavily skewed traffic. We pick 10 ToRs at random (25% of the ToRs). To skew the traffic, 80% of the flows have destinations set to hosts under these 10 ToRs. The remaining flows are routed to randomly chosen hosts. Figure 8a shows that the optimization is much more heavily impacted by the skew than 007. 007 continues to detect the cause of drops with high probability



(a) Single failure (b) Multiple failures

Figure 7: Varying the number of connections.



(a) Single failure (b) Multiple failures

Figure 8: 007's accuracy under skewed traffic.

($\geq 85\%$) for drop rates higher than 0.1%.

Multiple Failures. We repeated the above for multiple failures. Figure 8b shows that the optimization's accuracy suffers. It consistently shows a low detection rate as its constraints are not sufficient in guiding the optimizer to the right solution. 007 maintains a detection rate of $\geq 98\%$ at all times.

Hot ToR. A special instance of traffic skew occurs in the presence of a single hot ToR which acts as a sink for a large number of flows. Figure 9 shows how 007 performs in these situations. 007 can tolerate up to 50% skew, i.e., 50% of *all* flows go to the hot ToR, with negligible accuracy degradation. However, skews above 50% negatively impact its accuracy in the presence of a large number of failures (≥ 10). Such scenarios are unlikely as datacenter load balancing mitigates such extreme situations.

6.6 Detecting Bad Links

In our previous experiments, we focused on 007's accuracy on a per connection basis. In our next experiment, we evaluate its ability to detect bad links.

Single Failure. Figure 10 shows the results. 007

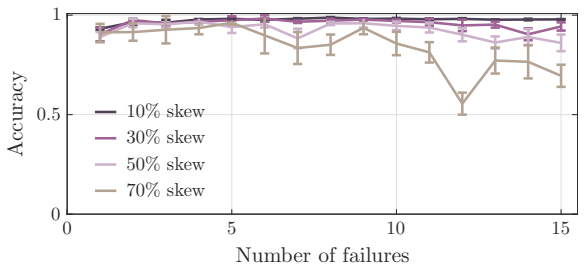
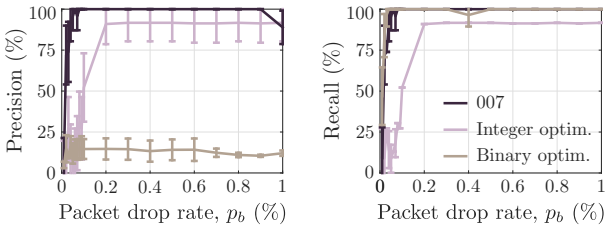
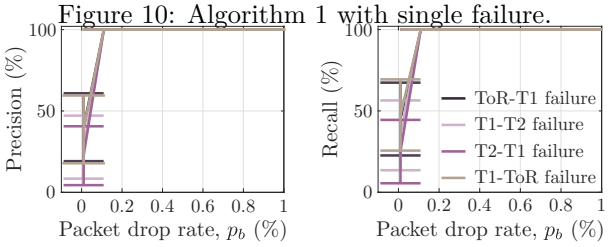


Figure 9: Impact of a hot ToR on 007's accuracy.



(a) (b)

Figure 10: Algorithm 1 with single failure.



(a) (b)

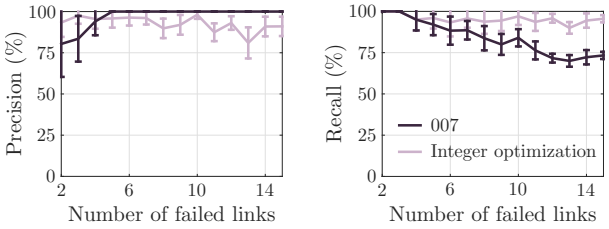
Figure 11: Impact of link location on Algorithm 1.

outperforms the optimization as it does not require a fully specified set of equations to provide a best guess as to which links failed. We also evaluate the impact of failure location on our results (Figure 11).

Multiple Failures. We heavily skew the drop rates on the failed links. Specifically, at least one failed link has a drop rate between 10 and 100%, while all others have a drop rate in (0.01%,0.1%). This scenario is one that past approaches have reported as hard to detect [2]. Figure 12 shows that 007 can detect up to 7 failures with accuracy above 90%. Its recall drops as the number of failed links increase. This is because the increase in the number of failures drives up the votes of all other links increasing the cutoff threshold and thus increasing the likelihood of false negatives. In fact if the top k links had been selected 007's recall would have been close to 100%.

6.7 Effects of Network Size

Finally, we evaluate 007 in larger networks. Its accuracy when finding a single failure was 98%, 92%, 91%, and 90% on average in a network with 1,2,3, and 4 pods respectively. In contrast, the optimization had an average accuracy of 94%, 72%, 79%, and 77% respectively. Algorithm 1 continues to have Recall



(a) (b)

Figure 12: Algorithm 1 with multiple failures. The drop rates on the links are heavily skewed.

$\geq 98\%$ for up to 6 pods (it drops to 85% for 7 pods). Precision remains 100% for all pod sizes.

We also evaluate both 007 and the optimization's ability to find the cause of per flow drops when the number of failed links is ≥ 30. We observe that both approach's performance remained unchanged for the most part, e.g., the accuracy of 007 in an example with 30 failed links is 98.01%.

7 Evaluations: Test Cluster

We next evaluate 007 on the more realistic environment of a test cluster with 10 ToRs and a total of 80 links. We control 50 hosts in the cluster, while others are production machines. Therefore, the T_1 switches see real production traffic. We recorded 6 hours of traffic from a host in production and replayed it from our hosts in the cluster (with different starting times). Using Everflow-like functionality [3] on the ToR switches, we induced different rates of drops on T_1 to ToR links. Our goal is to find the cause of packet drops on each flow §7.2 and to validate whether Algorithm 1 works in practice §7.3.

7.1 Clean Testbed Validation

We first validate a clean testbed environment. We repave the cluster by setting all devices to a clean state. We then run 007 without injecting any failures. We see that in the newly-repaved cluster, links arriving at a particular ToR switch had abnormally high votes, namely 22.5 ± 3.65 in average. We thus suspected that this ToR is experiencing problems. After rebooting it, the total votes of the links went down to 0, validating our suspicions. This exercise also provides one example of when 007 is extremely effective at identifying links with low drop rates.

7.2 Per-connection Failure Analysis

Can 007 identify the cause of drops when links have very different drop rates? To find out, we induce a drop rate of 0.2% and 0.05% on two different links for an hour. We only know the ground truth when the flow goes through at least one of the two failed links. Thus, we only consider such flows. For 90.47% of these, 007 was able to attribute the packet drop to the correct link (the one with higher drop rate).

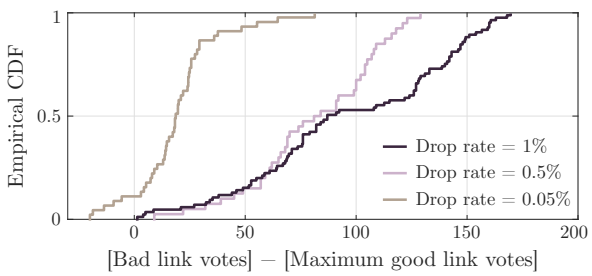


Figure 13: CDF of difference between votes on bad links and the maximum vote on good links.

7.3 Identifying Failed Links

We next validate Algorithm 1 and its ability to detect failed links. We inject different drop rates on a chosen link and determine whether there is a correlation between total votes and drop rates. Specifically, we look at the difference between the vote tally on the bad link and that of the most voted good link. We induced a packet drop rate of 1%, 0.1%, and 0.05% on a T_1 to ToR link in the test cluster.

Figure 13 shows the distribution for the various drop rates. The failed link has the highest vote out of all links when the drop rate is 1% and 0.1%. When the drop rate is lowered to 0.05%, the failed link becomes harder to detect due to the smaller gap between the drop rate of the bad link and that of the normal links. Indeed, the bad link only has the maximum score in 88.89% of the instances (mostly due to occasional lone drops on healthy links). However, it is always one of the 2 links with the highest votes.

Figure 13 also shows the high correlation between the probability of packet drop on a links and its vote tally. This trivially shows that 007 is 100% accurate in finding the cause of packet drops on each flow given a single link failure: the failed link has the highest votes among all links. We compare 007 with the optimization problem in (4). We find that the latter also returns the correct result every time, albeit at the cost of a large number of false positives. To illustrate this point: the number of links marked as bad by (4) on average is 1.5, 1.18, and 1.47 times higher than the number given by 007 for the drop rates of 1%, 0.1%, and 0.05% respectively.

What about multiple failures? This is a harder experiment to configure due to the smaller number of links in this test cluster and its lower path diversity. We induce different drop rates ($p_1 = 0.2\%$ and $p_2 = 0.1\%$) on two links in the cluster. The link with higher drop rate is the most voted 100% of the time. The second link is the second highest ranked 47% of the time and the third 32% of the time. It always remained among the 5 most voted links. This shows that by allowing a single false positive (identifying three instead of two links), 007 can detect all failed links 80% of the time even in a setup where the traffic distribution is highly skewed. This is something past approaches [2] could not achieve. In this example, 007 identifies the true cause of packet drops on each connection 98% of the time.

8 Evaluations: Production

We have deployed 007 in one of our datacenters[2]. Notable examples of problems 007 found include: power

[2]The monitoring agent has been deployed across all our data centers for over 2 years.

$T = 0$	$T > 0$ & $T \leq 3$	$T > 3$	$\max(T)$
69%	30.98%	0.02%	11

Table 1: Number of ICMPs per second per switch (T). We see $\max(T) \leq T_{\max}$.

supply undervoltages [28], FCS errors [29], switch reconfigurations, continuous BGP state changes, link flaps, and software bugs [30]. Also, 007 found every problem that was caught by our previously deployed diagnosis tools.

8.1 Validating Theorem 1

Table 1 shows the distribution of the number of ICMP messages sent by each switch in each epoch over a week. The number of ICMP messages generated by 007 never exceed T_{max} (Theorem 1).

8.2 TCP Connection Diagnosis

In addition to finding problematic links, 007 identifies the most likely cause of drops on each flow. Knowing when each individual packet is dropped in production is hard. We perform a semi-controlled experiment to test the accuracy of 007. Our environment consists of thousands of hosts/links. To find the "ground truth", we compare its results to that obtained by EverFlow. EverFlow captures all packets going through each switch on which it was enabled. It is expensive to run for extended periods of time. We thus only run EverFlow for 5 hours and configure it to capture all outgoing IP traffic from 9 random hosts. The captures for each host were conducted on different days. We filter all flows that were detected to have at least one retransmission during this time and using EverFlow find where their packets were dropped. We then check whether the detected link matches that found by 007. We found that *007 was accurate in every single case*. In this test we also verified that each path recorded by 007 matches exactly the path taken by that flow's packets as captured by EverFlow. This confirms that it is unlikely for paths to change fast enough to cause errors in 007's path discovery.

8.3 VM Reboot Diagnosis

During our deployment, there were 281 VM reboots in the datacenter for which there was no explanation. 007 found a link as the cause of problems in each case. Upon further investigation on the SNMP system logs, we observe that in 262 cases, there were transient drops on the host to ToR link a number of which were correlated with high CPU usage on the host. Two were due to high drop rates on the ToR. In another 15, the endpoints of the links found were undergoing configuration updates. In the remaining 2 instances, the link was flapping.

Finally, we looked at our data for one cluster for one day. 007 identifies an average of 0.45 ± 0.12 links as dropping packets per epoch. The average across all epochs of the maximum vote tally was 2.51 ± 0.33. Out of the links dropping packets 48% are server to ToR links (38% were due to a single ToR switch that was eventually taken out for repair), 24% are T_1-ToR links and 6% were due to T_2-T_1 link failures.

9 Discussion

007 is highly effective in finding the cause of packet drops on individual flows. By doing so, it provides flow-level context which is useful in finding the cause of problems for specific applications. In this section we discuss a number of other factors we considered in its design.

9.1 007's Main Assumptions

The proofs of Theorems 1 and 2 and the design of the path discovery agent (§4) are based on a number of assumptions:

ACK loss on reverse path. It is possible that packet loss on the reverse path is so severe that loss of ACK packets triggers timeout at the sender. If this happens, the traceroute would not be going over any link that triggered the packet drop. Since TCP ACKs are cumulative, this is typically not a problem and 007 assumes retransmissions in such cases are unlikely. This is true unless loss rates are very high, in which case the severity of the problem is such that the cause is apparent. Spurious retransmissions triggered by timeouts may also occur if there is sudden increased delay on forward or reverse paths. This can happen due to rerouting, or large queue buildups. 007 treats these retransmissions like any other.

Source NATs. Source network address translators (SNATs) change the source IP of a packet before it is sent out to a VIP. Our current implementation of 007 assumes connections are SNAT bypassed. However, if flows are SNATed, the ICMP messages will not have the right source address for 007 to get the response to its traceroutes. This can be fixed by a query to the SLB. Details are omitted.

L2 networks. Traceroute is not a viable option to find paths when datacenters operate using L2 routing. In such cases we recommend one of the following: (a) If access to the destination is not a problem and switches can be upgraded one can use the path discovery methods of [2, 31]. 007 is still useful as it allows for finding the cause of failures when multiple failures are present and for individual flows. (b) Alternatively, EverFlow can be used to find path. 007's sampling is necessary here as EverFlow doesn't scale to capture the path of all flows.

Network topology. The calculations in §5 assume a known topology (Clos). The same calculations can be carried out for *any* known topology by updating

the values used for ECMP. The accuracy of 007 is tied to the degree of path diversity and that multiple paths are available at each hop: the higher the degree of path diversity, the better 007 performs. This is also a desired property in any datacenter topology, most of which follow the Clos topology [13, 32, 33].

ICMP rate limit. In rare instances, the severity of a failure or the number of flows impacted by it may be such that it triggers 007's ICMP rate limit which stops sending more traceroute messages in that epoch. This *does not* impact the accuracy of Algorithm 1. By the time 007 reaches its rate limit, it has enough data to localize the problematic links. However, this limits 007's ability to find the cause of drops on flows for which it did not identify the path. We accept this trade-off in accuracy for the simplicity and lower overhead of 007.

Unpredictability of ECMP. If the topology and the ECMP functions on all the routers are known, the path of a packet can be found by inspecting its header. However, ECMP functions are typically proprietary and have initialization "seeds" that change with every reboot of the switch. Furthermore, ECMP functions change after link failures and recoveries. Tracking all link failures/recoveries in real time is not feasible at a datacenter scale.

9.2 Other Factors To Consider

007 has been designed for a specific use case, namely finding the cause of packet drops on individual connections in order to provide application context. This resulted in a number of design choices:

Detecting congestion. 007 *should not* avoid detecting major congestion events as they signal severe traffic imbalance and/or incast and are actionable. However, the more prevalent ($\geq 92\%$) forms of congestion have low drop rates 10^{-8}–10^{-5} [29]. 007 treats these as noise and does not detect them. Standard congestion control protocols can effectively react to them.

007's ranking. 007's ranking approach will naturally bias towards the detection of failed links that are frequently used. This is an intentional design choice as the goal of 007 is to identity high impact failures that affect many connections.

Finding the cause of other problems. 007's goal is to identify the cause of every packet drop, but other problems may also be of interest. 007 can be extended to identify the cause of many such problems. For example, for latency, ETW provides TCP's smooth RTT estimates upon each received ACK. Thresholding on these values allows for identifying "failed" flows and 007's voting scheme can be used to provides a ranked list of suspects. Proving the accuracy of 007 for such problems requires an extension of the analysis presented in this paper.

VM traffic problems. 007's goal is to find the cause of drops on infrastructure connections and through those, find the failed links in the network. In principle, we can build a 007-like system to diagnose TCP failures for connections established by customer VMs as well. For example, we can update the monitoring agent to capture VM TCP statistics through a VFP-like system [34]. However, such a system raises a number of new issues, chief among them being security. This is part of our future work.

In conclusion, we stress that the purpose of 007 is to *explain* the cause of drops when they occur. Many of these are not actionable and do not require operator intervention. The tally of votes on a given link provide a starting point for deciding when such intervention is needed.

10 Related Work

Finding the source of failures in distributed systems, specifically networks, is a mature topic. We outline some of the key differences of 007 with these works.

The most closely related work to ours is perhaps [2], which requires modifications to routers and both endpoints a limitation that 007 does not have. Often services (e.g. storage) are unwilling to incur the additional overhead of new monitoring software on their machines and in many instances the two endpoints are in seperate organizations [4]. Moreover, in order to apply their approach to our datacenter, a number of engineering problems need to be overcome, including finding a substitute for their use of the DSCP bit, which is used for other purposes in our datacenter. Lastly, while the statistical testing method used in [2] (as well as others) are useful when paths of both failed and non-failed flows are available they cannot be used in our setting as the limited number of traceroutes 007 can send prevent it from tracking the path of all flows. In addition 007 allows for diagnosis of individual connections *and* it works well in the presence of multiple simultaneous failures, features that [2] does not provide. Indeed, finding paths only when they are needed is one of the most attractive features of 007 as it minimizes its overhead on the system. Maximum cover algorithms [31, 35] suffer from many of the same limitations described earlier for the binary optimization, since MAX COVERAGE and Tomo are approximations of (3). Other related work can be loosely categorized as follows:

Inference and Trace-Based Algorithms [1, 2, 3, 36, 37, 38, 39, 40, 41, 42] use anomaly detection and trace-based algorithms to find sources of failures. They require knowledge/inference of the location of logical devices, e.g. load balancers in the connection

path. While this information is available to the network operators, it is not clear which instance of these entities a flow will go over. This reduces the accuracy of the results.

Everflow [3] aims to accurately identify the path of packets of interest. However, it does not scale to be used as an always on diagnosis system. Furthermore, it requires additional features to be enabled in the switch. Similarly, [26, 31] provides another means of path discovery, however, such approaches require deploying new applications to the remote end points which we want to avoid (due to reasons described in [4]). Also, they depend on SDN enabled networks and are not applicable to our setting where routing is based on BGP enabled switches.

Some inference approaches aim at *covering* the full topology, e.g. [1]. While this is useful, they typically only provides a sampled view of connection livelihood and do not achieve the type of always on monitoring that 007 provides. The time between probes for [1] for example is currently 5 minutes. It is likely that failures that happen at finer time scales slip through the cracks of its monitoring probes.

Other such work, e.g. [2, 39, 40, 41] require access to both endpoints and/or switches. Such access may not always be possible. Finally, NetPoirot [4] can only identify the general type of a problem (network, client, server) rather than the responsible device.

Network tomography [7, 8, 9, 11, 21, 43, 44, 45, 46, 47, 48, 49, 50] typically consist of two aspects: (i) the gathering and filtering of network traffic data to be used for identifying the points of failure [7, 45] and (ii) using the information found in the previous step to identify where/why failures occurred [8, 9, 10, 21, 43, 49, 51]. 007 utilizes ongoing traffic to detect problems, unlike these approaches which require a much heavier-weight operation of gathering large volumes of data. Tomography-based approaches are also better suited for non-transient failures, while 007 can handle both transient and persistent errors. 007 also has coverage that extends to the entire network infrastructure, and does not limit coverage to only paths between designated monitors as some such approaches do. Work on analyzing failures [7, 21, 43, 45] are complementary and can be applied to 007 to improve our accuracy.

Anomaly detection [52, 53, 54, 55, 56, 57, 58] find when a failure has occurred using machine learning [52, 54] and Fourier transforms [56]. 007 goes a step further by finding the device responsible.

Fault Localization by Consensus [59] assumes that a failure on a node common to the path used by a subset of clients will result in failures on a significant number of them. NetPoirot [4] illustrates why this approach fails in the face of a subset of problems that are common to datacenters. While our work builds on this idea, it provides a confidence measure that identifies how reliable a diagnosis report is.

Fault Localization using TCP statistics [2, 60, 61, 62, 63] use TCP metrics for diagnosis. [60] requires heavyweight active probing. [61] uses learning techniques. Both [61], and T-Rat [62] rely on continuous packet captures which doesn't scale. SNAP [63] identifies performance problems/causes for connections by acquiring TCP information which are gathered by querying socket options. It also gathers routing data combined with topology data to compare the TCP statistics for flows that share the same host, link, ToR, or aggregator switch. Given their lack of continuous monitoring, all of these approaches fail in detecting the type of problems 007 is designed to detect. Furthermore, the goal of 007 is more ambitious, namely to find the link that causes packet drops for each TCP connection.

Learning Based Approaches [4, 64, 65, 66] do failure detection in home and mobile networks. Our application domain is different.

Application diagnosis [67, 68] aim at identifying the cause of problems in a distributed application's execution path. The limitations of diagnosing network level paths and the complexities associated with this task are different. Obtaining all execution paths *seen* by an application, is plausible in such systems but is not an option in ours.

Failure resilience in datacenters [13, 69, 70, 71, 72, 73, 74, 75, 76, 77] target resilience to failures in datacenters. 007 can be helpful to a number of these algorithms as it can find problematic areas which these tools can then help avoid.

Understanding datacenter failures [22, 78] aims to find the various types of failures in datacenters. They are useful in understanding the types of problems that arise in practice and to ensure that our diagnosis engines are well equipped to find them. 007's analysis agent uses the findings of [22].

11 Conclusion

We introduced 007, an always on and scalable monitoring/diagnosis system for datacenters. 007 can accurately identify drop rates as low as 0.05% in datacenters with thousands of links through monitoring the status of ongoing TCP flows.

12 Acknowledgements

This work was was supported by grants NSF CNS-1513679, DARPA/I2O HR0011-15-C-0098. The authors would like to thank T. Adams, D. Dhariwal, A. Aditya, M. Ghobadi, O. Alipourfard, A. Haeberlen, J. Cao, I. Menache, S. Saroiu, and our shepherd H. Madhyastha for their help.

References

[1] 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), pp. 139–152.

[2] Roy, A., Bagga, J., Zeng, H., and Sneoren, A. Passive realtime datacenter fault detection. In *ACM NSDI* (2017).

[3] 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 *ACM SIGCOMM* (2015), pp. 479–491.

[4] Arzani, B., Ciraci, S., Loo, B. T., Schuster, A., Outhred, G., et al. Taking the blame game out of data centers operations with NetPoirot. In *ACM SIGCOMM* (2016), pp. 440–453.

[5] Wu, X., Turner, D., Chen, C.-C., Maltz, D. A., Yang, X., Yuan, L., and Zhang, M. NetPilot: Automating datacenter network failure mitigation. *ACM SIGCOMM Computer Communication Review 42*, 4 (2012), 419–430.

[6] Narayana, S., Sivaraman, A., Nathan, V., Goyal, P., Arun, V., Alizadeh, M., Jeyakumar, V., and Kim, C. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 85–98.

[7] Zhang, Y., Roughan, M., Willinger, W., and Qiu, L. Spatio-temporal compressive sensing and internet traffic matrices. *ACM SIGCOMM Computer Communication Review 39*, 4 (2009), 267–278.

[8] Ma, L., He, T., Swami, A., Towsley, D., Leung, K. K., and Lowe, J. Node failure localization via network tomography. In *ACM SIGCOMM IMC* (2014), pp. 195–208.

[9] Liu, C., He, T., Swami, A., Towsley, D., Salonidis, T., and Leung, K. K. Measurement design framework for network tomography using fisher information. *ITA AFM* (2013).

[10] Dhamdhere, A., Teixeira, R., Dovrolis, C., and Diot, C. NetDiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *ACM CoNEXT* (2007).

[11] Kompella, R. R., Yates, J., Greenberg, A., and Snoeren, A. C. IP fault localization via risk modeling. In *USENIX NSDI* (2005), pp. 57–70.

[12] Bertsimas, D., and Tsitsiklis, J. N. *Introduction to linear optimization*. Athena Scientific, 1997.

[13] Ports, D. R. K., Li, J., Liu, V., Sharma, N. K., and Krishnamurthy, A. Designing distributed systems using approximate synchrony in data center networks. In *USENIX NSDI* (2015), pp. 43–57.

[14] Microsoft. Windows ETW, 2000. `https://msdn.microsoft.com/en-us/library/windows/desktop/bb968803(v=vs.85).aspx`.

[15] Hopps, C. E. RFC 2992: Analysis of an Equal-Cost Multi-Path algorithm, 2000.

[16] 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. *ACM SIGCOMM Computer Communication Review 43*, 4 (2013), 207–218.

[17] Liu, H. H., Kandula, S., Mahajan, R., Zhang, M., and Gelernter, D. Traffic engineering with forward fault correction. In *ACM SIGCOMM Computer Communication Review* (2014), vol. 44, ACM, pp. 527–538.

[18] Johnson, B. W., Kim, S. H., Leo Jr, E. J., and Lee, D. Link aggregation path selection method, 2003. US Patent 6,535,504.

[19] Gunes, M. H., and Sarac, K. Resolving IP aliases in building traceroute-based internet maps. *IEEE/ACM Transactions on Networking 17*, 6 (2009), 1738–1751.

[20] Institute, I. S. RFC 791: Internet Protocol, 1981. DARPA.

[21] Mysore, R. N., Mahajan, R., Vahdat, A., and Varghese, G. Gestalt: Fast, unified fault localization for networked systems. In *USENIX ATC* (2014), pp. 255–267.

[22] Zhuo, D., Ghobadi, M., Mahajan, R., Phanishayee, A., Zou, X. K., Guan, H., Krishnamurthy, A., and Anderson, T. RAIL: A case for Redundant Arrays of Inexpensive Links in data center networks. In *USENIX NSDI* (2017).

[23] Bernhard, K., and Vygen, J. Combinatorial optimization: Theory and algorithms. *Springer, Third Edition, 2005*. (2008).

[24] Mosek, A. The mosek optimization software. *Online at http://www. mosek. com 54* (2010), 2–1.

[25] Arzani, B. Simulation source codes. Tech. rep., Microsoft Research, 2018. `https://github.com/behnazak/Vigil-007SourceCode.git`.

[26] Tammana, P., Agarwal, R., and Lee, M. Cherrypick: Tracing packet trajectory in software-defined datacenter networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (2015), ACM, p. 23.

[27] Arzani, B., Ciraci, S., Chamon, L., Zhu, Y., Liu, H., Padhye, J., Thau Loo, B., and Outhred, G. 007: Democratically finding the cause of packet drops. *arXiv preprint* (2018).

[28] Arista. Arista eos system message guide. Tech. rep., Arista Networks, 2015. `http://simatinc.com/simatftp/4.14/EOS-4.14.6M/EOS-4.14.6M-SysMsgGuide.pdf`.

[29] Zhuo, D., Ghobadi, M., Mahajan, R., Förster, K.-T., Krishnamurthy, A., and Anderson, T. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 362–375.

[30] Cisco. Cisco bug: Cscut86141 - sfp-h10gb-cu2.255m, hardware type changed to no-transceiver on n3k. Tech. rep., Cisco, 2017. `https://quickview.cloudapps.cisco.com/quickview/bug/CSCut86141`.

[31] Tammana, P., Agarwal, R., and Lee, M. Simplifying datacenter network debugging with pathdump. In *OSDI* (2016), pp. 233–248.

[32] Al-Fares, M., Loukissas, A., and Vahdat, A. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review* (2008), vol. 38, ACM, pp. 63–74.

[33] 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 *ACM SIGCOMM computer communication review* (2009), vol. 39, ACM, pp. 51–62.

[34] Firestone, D. Vfp: A virtual switch platform for host sdn in the public cloud. In *NSDI* (2017), pp. 315–328.

[35] Kompella, R. R., Yates, J., Greenberg, A., and Snoeren, A. C. Detection and localization of network black holes. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE* (2007), IEEE, pp. 2180–2188.

[36] 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. *ACM SIGCOMM Computer Communication Review 37*, 4 (2007), 13–24.

[37] Adair, K. L., Levis, A. P., and Hruska, S. I. Expert network development environment for automating machine fault diagnosis. In *SPIE Applications and Science of Artificial Neural Networks* (1996), pp. 506–515.

[38] Ghasemi, M., Benson, T., and Rexford, J. RINC: Real-time Inference-based Network diagnosis in the Cloud. Tech. rep., Princeton University, 2015. `https://www.cs.princeton.edu/research/techreps/TR-975-14`.

[39] Mahajan, R., Spring, N., Wetherall, D., and Anderson, T. User-level internet path diagnosis. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 106–119.

[40] Liú, Y., Miao, R., Kim, C., and Yuú, M. LossRadar: Fast detection of lost packets in data center networks. In *ACM CoNEXT* (2016), pp. 481–495.

[41] Li, Y., Miao, R., Kim, C., and Yu, M. FlowRadar: A better NetFlow for data centers. In *USENIX NSDI* (2016), pp. 311–324.

[42] Heller, B., Scott, C., McKeown, N., Shenker, S., Wundsam, A., Zeng, H., Whitlock, S., Jeyakumar, V., Handigol, N., McCauley, J., et al. Leveraging SDN layering to systematically troubleshoot networks. In *ACM SIGCOMM HotSDN* (2013), pp. 37–42.

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

[44] Kandula, S., Katabi, D., and Vasseur, J.-P. Shrink: A tool for failure diagnosis in IP networks. In *ACM SIGCOMM MineNet* (2005), pp. 173–178.

[45] Ogino, N., Kitahara, T., Arakawa, S., Hasegawa, G., and Murata, M. Decentralized boolean network tomography based on network partitioning. In *IEEE/IFIP NOMS* (2016), pp. 162–170.

[46] Chen, Y., Bindel, D., Song, H., and Katz, R. H. An algebraic approach to practical and scalable overlay network monitoring. *ACM SIGCOMM Computer Communication Review 34*, 4 (2004), 55–66.

[47] Zhao, Y., Chen, Y., and Bindel, D. Towards unbiased end-to-end network diagnosis. *ACM SIGCOMM Computer Communication Review 36*, 4 (2006), 219–230.

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

[49] Duffield, N. G., Arya, V., Bellino, R., Friedman, T., Horowitz, J., Towsley, D., and Turletti, T. Network tomography from aggregate loss reports. *Performance Evaluation 62*, 1 (2005), 147–163.

[50] Herodotou, H., Ding, B., Balakrishnan, S., Outhred, G., and Fitter, P. Scalable near real-time failure localization of data center networks. In *ACM KDD* (2014), pp. 1689–1698.

[51] Banerjee, D., Madduri, V., and Srivatsa, M. A framework for distributed monitoring and root cause analysis for large IP networks. In *IEEE SRDS* (2009), pp. 246–255.

[52] Fu, Q., Lou, J.-G., Wang, Y., and Li, J. Execution anomaly detection in distributed systems through unstructured log analysis. In *IEEE ICDM* (2009), pp. 149–158.

[53] Huang, L., Nguyen, X., Garofalakis, M., Jordan, M. I., Joseph, A., and Taft, N. In-network PCA and anomaly detection. In *NIPS* (2006), pp. 617–624.

[54] Gabel, M., Sato, K., Keren, D., Matsuoka, S., and Schuster, A. Latent fault detection with unbalanced workloads. In *EPForDM* (2015).

[55] Ibidunmoye, O., Hernández-Rodriguez, F., and Elmroth, E. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys 48*, 1 (2015).

[56] Zhang, Y., Ge, Z., Greenberg, A., and Roughan, M. Network anomography. In *ACM SIGCOMM IMC* (2005).

[57] Crovella, M., and Lakhina, A. Method and apparatus for whole-network anomaly diagnosis and method to detect and classify network anomalies using traffic feature distributions, 2014. US Patent 8,869,276.

[58] Kind, A., Stoecklin, M. P., and Dimitropoulos, X. Histogram-based traffic anomaly detection. *IEEE Transactions on Network and Service Management 6*, 2 (2009).

[59] Padmanabhan, V. N., Ramabhadran, S., and Padhye, J. Netprofiler: Profiling wide-area networks using peer cooperation. In *IPTPS*. 2005, pp. 80–92.

[60] Mathis, M., Heffner, J., O'Neil, P., and Siemsen, P. Pathdiag: Automated TCP diagnosis. In *PAM* (2008), pp. 152–161.

[61] Widanapathirana, C., Li, J., Sekercioglu, Y. A., Ivanovich, M., and Fitzpatrick, P. Intelligent automated diagnosis of client device bottlenecks in private clouds. In *IEEE UCC* (2011), pp. 261–266.

[62] Zhang, Y., Breslau, L., Paxson, V., and Shenker, S. On the characteristics and origins of internet flow rates. *ACM SIGCOMM Computer Communication Review 32*, 4 (2002), 309–322.

[63] Yu, M., Greenberg, A. G., Maltz, D. A., Rexford, J., Yuan, L., Kandula, S., and Kim, C. Profiling network performance for multi-tier data center applications. In *USENIX NSDI* (2011).

[64] Chen, M., Zheng, A. X., Lloyd, J., Jordan, M., Brewer, E., et al. Failure diagnosis using decision trees. In *IEEE ICAC* (2004), pp. 36–43.

[65] Dimopoulos, G., Leontiadis, I., Barlet-Ros, P., Papagiannaki, K., and Steenkiste, P. Identifying the root cause of video streaming issues on mobile devices.

[66] Agarwal, B., Bhagwan, R., Das, T., Eswaran, S., Padmanabhan, V. N., and Voelker, G. M. NetPrints: Diagnosing home network misconfigurations using shared knowledge. In *USENIX NSDI* (2009), vol. 9, pp. 349–364.

[67] Chen, Y.-Y. M., Accardi, A., Kiciman, E., Patterson, D. A., Fox, A., and Brewer, E. A. Path-based failure and evolution management. In *USENIX NSDI* (2004).

[68] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 74–89.

[69] Liu, J., Panda, A., Singla, A., Godfrey, B., Schapira, M., and Shenker, S. Ensuring connectivity via data plane mechanisms. In *USENIX NSDI* (2013), pp. 113–126.

[70] Alizadeh, M., Edsall, T., Dharmapurikar, S., Vaidyanathan, R., Chu, K., Fingerhut, A., Matus, F., Pan, R., Yadav, N., Varghese, G., et al. CONGA: Distributed congestion-aware load balancing for datacenters. *ACM SIGCOMM Computer Communication Review 44*, 4 (2014), 503–514.

[71] Paasch, C., and Bonaventure, O. Multipath TCP. *Communications of the ACM 57*, 4 (2014), 51–57.

[72] Chen, G., Lu, Y., Meng, Y., Li, B., Tan, K., Pei, D., Cheng, P., Luo, L. L., Xiong, Y., Wang, X., et al. Fast and cautious: Leveraging multi-path diversity for transport loss recovery in data centers. In *USENIX ATC* (2016).

[73] Schiff, L., Schmid, S., and Canini, M. Ground control to major faults: Towards a fault tolerant and adaptive SDN control network. In *IEEE/IFIP DSN* (2016), pp. 90–96.

[74] Reitblatt, M., Canini, M., Guha, A., and Foster, N. Fattire: Declarative fault tolerance for software-defined networks. In *ACM SIGCOMM HotSDN* (2013), pp. 109–114.

[75] Kuźniar, M., Perešíni, P., Vasić, N., Canini, M., and Kostić, D. Automatic failure recovery for software-defined networks. In *ACM SIGCOMM HotSDN* (2013), pp. 159–160.

[76] Bodík, P., Menache, I., Chowdhury, M., Mani, P., Maltz, D. A., and Stoica, I. Surviving failures in bandwidth-constrained datacenters. In *ACM SIGCOMM* (2012), pp. 431–442.

[77] Wundsam, A., Mehmood, A., Feldmann, A., and Maennel, O. Network troubleshooting with mirror VNets. In *IEEE GLOBECOM* (2010), pp. 283–287.

[78] Gill, P., Jain, N., and Nagappan, N. Understanding network failures in data centers: Measurement, analysis, and implications. *ACM SIGCOMM Computer Communication Review 41*, 4 (2011), 350–361.

[79] Arratia, R., and Gordon, L. Tutorial on large deviations for the binomial distribution. *Bulletin of Mathematical Biology 51*, 1 (1989), 125–131.

[80] Feller, W. *An Introduction to Probability Theory and Its Applications*, vol. 1. Wiley, 1968.

[81] Cover, T., and Thomas, J. *Elements of information theory*. Wiley-Interscience, 2006.

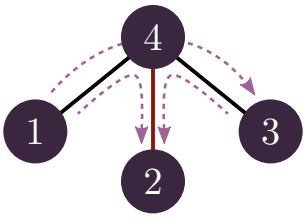


Figure 14: Simple tomography example.

A Network tomography example

Knowing the path of all flows, it is possible to find with confidence which link dropped a packet. To do so, consider the example network in Figure 14. Suppose that the link between nodes 2 and 4 drops packets. Flows 1–2 and 3–2 suffer from drops, but 1–3 does not. A set cover optimization, such as the one used by MAX COVERAGE and Tomo [10, 11], that minimizes the number of "blamed" links will correctly find the cause of drops. This problem is however equivalent to a set covering optimization problem that is known to be NP-complete [23].

B Proofs

Consider a Clos topology with n_{pod} pods each with n_0 ToR switches each with H hosts. Links between tier-0 and tier-1 switches are referred to as *level 1 links* and links between tier-1 and tier-2 switches are called *level 2 links* (see Figure 15). Assume that connection occur uniformly at random between hosts under different ToR switches. Also, assume that link failure and connection routing are independent and that links drop packets independently across links and across packets. To make the derivations clearer, we use calligraphic letter ($\mathcal{A}$) for sets and boldface ($\boldsymbol{A}$) to denote random variables. We also use the notation $[M] = 1, \ldots, M$ (see [27] for more detailed proofs).

B.1 Proof of Theorem 1

Proof. First, note that since the number of hosts below each ToR switch is the same, we can consider that traceroutes are sent uniformly at random between ToR switches at a rate $C_t H$. Also, note that routing probabilities are the same for links on the same level, so the traceroute rate depends only on whether the link is on level 1 or level 2. Given ECMP, the traceroute rates on a level 1 (R_1) and level 2 (R_2) link is given by

$$R_1 = \frac{1}{n_1} C_t H \text{ and } R_2 = \frac{n_0}{n_1 n_2} \frac{n_0(n_{\text{pod}}-1)}{(n_0 n_{\text{pod}}-1)} C_t H.$$

Since n_0 links are connected to a tier-1 switch and n_1 links are connected to a tier-2, the rate of ICMP packets at any links is bounded by $T \leq \max[n_0 R_1, n_1 R_2]$. Taking $\max[n_0 R_1, n_1 R_2] \leq T_{\max}$ yields (1). □

B.2 Proof of Theorem 2

We prove a more precise statement of Theorem 2.

Theorem 3. *In a Clos topology with $n_0 \geq n_2$ and $n_{\text{pod}} \geq 1 + \max\left[\frac{n_0}{n_1}, \frac{n_2(n_0-1)}{n_0(n_0-n_2)}, 1\right]$, 007 will rank with probability $(1-\epsilon)$ the $k < \frac{n_2(n_0 n_{\text{pod}}-1)}{n_0(n_{\text{pod}}-1)}$ bad links that drop packets with probability p_b above all good links that drop packets with probability p_g as long as*

$$p_g \leq \frac{1-(1-p_b)^{c_l}}{\alpha c_u}, \tag{5}$$

where c_l and c_u are lower and upper bounds, respectively, on the number of packets per connection, $\epsilon \leq 2e^{-\mathcal{O}(N)}$, N is the total number of connections between hosts, and

$$\alpha = \frac{n_0(4n_0-k)(n_{\text{pod}}-1)}{n_2(n_0 n_{\text{pod}}-1) - n_0(n_{\text{pod}}-1)k}. \tag{6}$$

Before proceeding, note that the typical scenario in which $n_0 \geq 2n_2$ and $\frac{n_2(n_0-1)}{n_0(n_0-n_2)} \leq 1$, as in our data center, the condition on the number of pods from Theorem 3 reduces to $n_{\text{pod}} \geq 1 + \frac{n_0}{n_1}$.

Proof. The proof proceeds as follows. First, we show that if a link has higher probability of receiving a vote, then it receives more votes if a large enough number of connections (N) are established. We do so using large deviation theory [79], so that we can show that the probability that this does not happen decreases exponentially in N.

Lemma 1. *Let v_b (v_g) be the probability of a bad (good) link receiving a vote. If $v_b \geq v_g$, 007 ranks bad links above good links with probability $1 - e^{-\mathcal{O}(N)}$.*

With Lemma 1 in hands, we then need to relate the probabilities of a link receiving a vote (v_b, v_g) to the link drop rates (p_b, p_g). This will allow us to derive the signal-to-noise ratio condition in (5). Note that the probability of a link receiving a vote is the probability of a flow going through the link *and* that a retransmission occurs (i.e., some link in the flow's path drops at least one packet).

Lemma 2. *In a Clos topology with $n_0 \geq n_2$ and $n_{\text{pod}} \geq 1 + \max\left[\frac{n_0}{n_1}, \frac{n_2(n_0-1)}{n_0(n_0-n_2)}, 1\right]$, it holds that for $k \leq n_0$ bad links*

$$v_b \geq \frac{r_b}{n_0 n_1 n_{\text{pod}}} \tag{7a}$$

$$v_g \leq \frac{1}{n_1 n_2 n_{\text{pod}}} \frac{n_0(n_{\text{pod}}-1)}{n_0 n_{\text{pod}} - 1} \left[(4 - \frac{k}{n_0}) r_g + \frac{k}{n_0} r_b\right] \tag{7b}$$

where r_b and r_g are the probabilities of a retransmission being due to a bad and a good link, respectively.

Before proving these lemmata, let us see how they imply Theorem 3. From (7) in Lemma 2,

$$r_b \geq \underbrace{\frac{n_0(4n_0-k)(n_{\text{pod}}-1)}{n_2(n_0 n_{\text{pod}}-1)-n_0(n_{\text{pod}}-1)k}}_{\alpha} r_g \Rightarrow v_b \geq v_g, \tag{8}$$

for $k < \frac{n_2(n_0 n_{\text{pod}}-1)}{n_0(n_{\text{pod}}-1)} < n_0$. Thus, in a Clos topology, if $r_b \geq \alpha r_g$ for α as in (6), then $v_b \geq v_g$. However, (8) gives a relation in terms of probabilities of retransmission (r_g, r_b) instead of the packet drop rates (p_g, p_b). Nevertheless, note that the probability r of retransmission during a connection with c packets due to a link that drops packets with probability p is $r = 1-(1-p)^c$. Since r is monotonically increasing in c, we have $r_b \geq 1-(1-p_b)^{c_l}$ and $r_g \leq 1-(1-p_g)^{c_u}$. Using the fact $(1-x)^n \geq 1-nx$ yields (5). □

Proof of Lemma 1. First, note in a datacenter-sized Clos network, almost every connection has a hop count of 5 (in our datacenter, this happens to 97.5% of connections). We can therefore approximate links votes by assuming all bad votes have the same value.

Since links cause retransmissions independently across connections, the number of votes of a bad link is a binomial random variable $\boldsymbol{B}$ with parameters N, the total number of connections, and v_b, the probability of a bad link receiving a bad vote. Similarly, let $\boldsymbol{G}$ be the number of votes on a good link, a binomial random variable with parameters N and v_g. 007 will correctly rank bad links if $\boldsymbol{B} \geq \boldsymbol{G}$, i.e., if bad links receive more votes than good links. This event contains the event $\mathcal{D} = \{\boldsymbol{G} \leq (1+\delta)Nv_g \cap \boldsymbol{B} \geq (1-\delta)Nv_b\}$ for $\delta \leq \frac{v_b - v_g}{v_b + v_g}$. Using the union bound [80], the probability of 007 correctly identifying bad links obeys

$$\begin{aligned}\mathbb{P}(\boldsymbol{B} \geq \boldsymbol{G}) \geq 1 &- \mathbb{P}[\boldsymbol{G} \geq (1+\delta)Nv_g] \\ &- \mathbb{P}[\boldsymbol{B} \leq (1-\delta)Nv_b].\end{aligned} \tag{9}$$

To proceed, bound the probabilities in (9) using the large deviation principle [79], i.e., use the fact that for a binomial random variable $\boldsymbol{S}$ with parameters M and q and for $\delta > 0$ it holds that

$$\mathbb{P}[\boldsymbol{S} \geq (1+\delta)qM] \leq e^{-M\,\mathrm{D}_{\mathrm{KL}}((1+\delta)q \| q)} \tag{10a}$$

$$\mathbb{P}[\boldsymbol{S} \leq (1-\delta)qM] \leq e^{-M\,\mathrm{D}_{\mathrm{KL}}((1-\delta)q \| q)} \tag{10b}$$

where $\mathrm{D}_{\mathrm{KL}}(q\|r)$ is the Kullback-Leibler divergence between two Bernoulli distributions with probabilities of success q and r [81]. The result in Lemma 1 is obtained by substituting (10) into (9). □

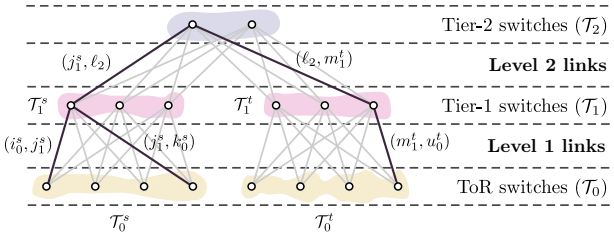


Figure 15: Illustration of notation for Clos topology used in the proof of Lemma 2

Proof of Lemma 2. Let $\mathcal{T}_0$, $\mathcal{T}_1$, and $\mathcal{T}_2$ denote the set of ToR, tier-1, and tier-2 switches respectively (Figure 15). Let $\mathcal{T}_0^s$ and $\mathcal{T}_1^s$, $s = [n_{\text{pod}}]$, denote the tier-0 and tier-1 switches in pod s, respectively. Note that $\mathcal{T}_0 = \mathcal{T}_0^1 \cup \cdots \cup \mathcal{T}_0^{n_{\text{pod}}}$ and $\mathcal{T}_1 = \mathcal{T}_1^1 \cup \cdots \cup \mathcal{T}_1^{n_{\text{pod}}}$. Throughout the derivation, we use subscripts to denote the switch tier and superscripts to denote its pod. For instance, i_0^s is the i-th tier-0 switch from pod s, i.e., $i_0^s \in \mathcal{T}_0^s$, and ℓ_2 is the ℓ-th tier-2 switch (tier-2 switches do not belong to specific pods). We write (i_0^s, j_1^s) for the level 1 link between i_0^s to j_1^s (as in Figure 15) and $r(i_0^s, j_1^s) = r(j_1^s, i_0^s)$ to refer to the probability of link (i_0^s, j_1^s) causing a retransmission.

To get the lower bound in (7a), note that a bad link receives at least as many bad votes as retransmissions it causes. Therefore, the probability of 007 voting for a bad link is larger than the probability of that link causing a retransmission. The bound is obtained by taking the minimum between the probability of a connection going through a level 1 and a level 2 link and that link causing a retransmission, i.e.,

$$v_b \geq \min\left[\frac{1}{n_0 n_1 n_{\text{pod}}}, \frac{1}{n_1 n_2 n_{\text{pod}}} \frac{n_0(n_{\text{pod}}-1)}{n_0 n_{\text{pod}} - 1}\right] r_b.$$

The assumption that $n_{\text{pod}} \geq 1 + \frac{n_2(n_0-1)}{n_0(n_0-n_2)}$ makes the first term smaller than the second and yields (7a).

In contrast, the upper bound in (7b) is obtained by applying the union bound [80] over all possible ways that level 1 and level 2 link could be voted. Then, finding an adversarial placement of good and bad links that maximizes the probability of the good link receiving a vote, we find that for $n_0 \geq n_2$, $n_{\text{pod}} \geq 2$, and $k \leq n_2$, the worst case is achieved for a level 2 link, yielding (7a). □