Grasp the Root Causes in the Data Plane: Diagnosing Latency Problems with SpiderMon

Weitao Wang
Rice University

Praveen Tammana
Princeton University

Ang Chen
Rice University

T. S. Eugene Ng
Rice University

ABSTRACT

Unexplained performance degradation is one of the most severe problems in data center networks. The increasing scale of the network makes it even harder to maintain good performance for all users with a low-cost solution. Our system SpiderMon monitors network performance and debugs performance failures inside the network with little overhead. SpiderMon provides a two-phase solution that runs in the data plane. In the monitoring phase, it keeps track of the performance of every flow in the network; upon detecting performance problems, it triggers a debugging phase using a causality analyzer to find out the root cause of performance degradation. To implement these two phases, SpiderMon exploits the capabilities of high-speed programmable switches (*e.g.*, per-packet monitoring, stateful memory). We prototype SpiderMon on using the `BMv2` model of P4, and our preliminary evaluation shows that SpiderMon is able to quickly find the root cause of performance degradation problems with minimal overhead. SpiderMon achieves nearly-zero overhead during the monitoring phase and efficiently collects relevant data from switches during the debugging phase.

CCS CONCEPTS

• **Networks** **Network monitoring**; **Programmable networks**; **Data center networks**.

KEYWORDS

Performance diagnosis, in-network telemetry, P4, network provenance

1 INTRODUCTION

A low-cost network diagnostic system is essential to meeting performance requirements of modern applications. Many performance degradation problems are caused by traffic contention [5], and such contention can lead to high end-to-end delays for both related and unrelated traffic [31]. Therefore, it is critical to monitor performance by collecting fine-grained information and process the information to pinpoint the root causes of performance degradation. By doing so, network operators can understand their networks better and use appropriate configurations to meet the performance requirements. However, as the network size grows, collecting and processing the information for diagnosis become extremely expensive and challenging.

Broadly, the task of performance diagnosis can be divided into monitoring and debugging phases. In the monitoring phase, a system needs to *detect* high end-to-end delay that traffic may experience. In the debugging phase, the system should identify the *root cause* for the abnormally high delay. To do this, each phase requires different types of information at different locations in the network. For instance, consider a problem where packets experienced high end-to-end delay due to traffic contention (therefore queuing delay) across multiple hops. Detecting the delay would require tracking the time that packets spent at each hop; further identifying the root cause would require tracking flows that shared the same queues with these packets. Moreover, such information needs to be collected in a network-wide manner.

There has been much recent work on network diagnosis. On the one hand, we have systems that run either at hosts or switches [6, 9, 11, 18, 20, 21] which leverage programmable switches to monitor traffic and collect fine-grained information (*e.g.*, flow-level, packet-level) on small time scales (*e.g.*, milliseconds to seconds). This design choice enables high network-wide visibility, but incurs a large resource overheads (*e.g.*, network bandwidth, processing, and storage). Alternatively, query-based systems [10, 13, 25, 26, 32, 34, 35] reduce the overhead by executing a set of diagnostic queries on packet streams and filter the relevant data. However, these systems do not keep track of the flows that share the queues across switches, thus cannot do network-wide diagnosis accurately. On the other hand, systems relying on both switches and hosts [7, 12, 14, 16, 19, 23, 24, 29–31, 36] for network-wide diagnosis leverage the resources at the hosts to collect historical data and maintain flow-level statistics. But, they tend to be too slow to react to "gray failures" (*e.g.*, performance degradation) as the problem might disappear by the time the hosts detect it, inform a controller, and the controller retrieves data; thus, they are inaccurate.

Therefore, having a diagnosis system that achieves either high accuracy or low overhead is not hard, but achieving both simultaneously is challenging.

We present SpiderMon, a network-wide diagnosis system that aims to bridge the gap between accuracy and overhead by monitoring and collecting relevant telemetry data in a distributed manner. The key idea is that every switch maintains fine-grained telemetry data (*e.g.*, per-flow records) in the data plane for a short period of time depending on the available memory resources, and the information is offloaded to a central entity only when a performance degradation

SOSR '20, March 3, 2020, San Jose, CA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7101-8/20/03…$15.00
https://doi.org/10.1145/3373360.3380835

(*e.g.*, high latency) is detected. In this way, the central entity would receive only a tiny fraction of network-wide telemetry data while still be able to accurately find the root cause of the performance problem by correlating the telemetry data received from a small subset of relevant switches.

To realize this idea in practice, SpiderMon resolves two technical challenges. The first challenge is to detect the performance degradation without interfering with the actual packet processing. For this we leverage a capability of programmable switches that provides the amount of time a packet is spent in a queue. SpiderMon piggybacks the accumulated delay information in every packet header, and checks whether the delay exceeds a certain threshold at every hop. If so, a problem is detected (more details in §3.1).

The second challenge is to debug and find the root cause of the performance degradation. For this, SpiderMon notifies and offloads telemetry data (*e.g.*, per-flow records) relevant to the degradation from the involved switches in the network. SpiderMon views this as a *provenance graph* of the network events that are related to the performance degradation. The abstraction of the provenance graph captures all the events which cause the degradation as nodes in the graph, and the causalities among events (*e.g.*, flow contentions) are represented using edges in the graph (more details in §2.2).

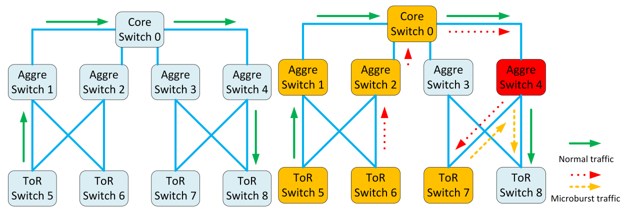
To notify relevant switches in the graph, SpiderMon provides an audit request system using stateful memory in the programmable switches. The system maintains two compact data structures: (1) a per-switch timeout bloom filter that keeps track of flows-to-port mappings; (2) a per-port per-epoch data structure that keeps track of the incoming ports on which traffic is received (more details in §3.3). When SpiderMon detects a performance degradation, the system issues a notification (*i.e.*, audit request) and uses these two data structures to propagate the audit request to relevant switches in the network. Every switch that receives the audit request would offload its local telemetry data (*e.g.*, flow records) to a central monitoring server for analysis. For instance, to find the root cause, we can construct a flow-level provenance graph and find the root cause by correlating the flow-level information in both temporal and spatial dimensions.

Contributions. We present SpiderMon, a *lightweight* system to diagnose latency problems *accurately*. We have implemented an initial prototype of SpiderMon, and our preliminary results show that SpiderMon can diagnose latency problems with high accuracy while consuming minimal switch memory (tens of KBs) and control plane bandwidth (tens of Mbps).

2 OVERVIEW

2.1 Network Performance Degradation

As a concrete example, consider the case presented in Figs.1(a) and 1(b). The green flow shows a victim TCP connection which is forwarded from switch 5 to switch 8 through switches 1, 0 and 4. In the middle of the transfer, two UDP flows start transferring from switch 6 to 7 and from 7 to 8 separately. From this time, the green TCP flow will get delayed at switch 0, then switch 4, and the accumulated delay would exceed an acceptable threshold at switch 4. Here, the high end-to-end delay is the accumulated result of multiple smaller delays along the victim flow's path, so it requires information from all the switches along that path for



(a) No conflict (b) Accumulated conflict

Figure 1: Multiple contentions have caused a transient performance problem

a successful diagnosis. Besides, such performance problems can be sporadic, because they do not deterministically depend on the network configuration. Because multiple contention happening at the same time is not a high-probability event, and a contention only lasts for a short amount of time, these problems are also transient in their appearance.

As we can see, network performance degradation problems are a kind of "gray failures", which are subtle to detect and diagnose but can cause significant problems to the applications, such as contention between multiple flows, priority contention, and load imbalance. There are three common key features that make these problems challenging to diagnose.

Sporadic. Performance degradation are usually sporadic—i.e., they happen occasionally at different places and at unpredictable times [3]. In order to detect these problems, an effective solution needs to monitor every flow all the time.

Network-wide. The root causes for complex performance problems' may be network-wide, e.g., due to the contention of multiple flows at different hops. The interfering flows may even have normal performance [31], despite the fact that they cause performance degradation to other flows. Thus root cause diagnosis requires network-wide monitoring.

Transient. Traffic contentions sometimes are transient and disappear quickly [17], because degradation happens when there are multiple flows contending for resources. This feature requires the debugging system to keep fine-grained information about recent events.

2.2 The Provenance Model

To diagnose network performance problems in a sufficient and efficient way, we need flow-level information from all switches that are causally related. Determining which types of information are relevant can be guided by the following provenance model.

We define the provenance graph model $G := \langle V; E \rangle$ for all events in the network. G is a directed acyclic graph, where each node v represents an event, and each directed edge $e = \langle v_1 \rightarrow v_2 \rangle$ between two nodes represents the provenance relation that v_1 leads to the event v_2. One node can have multiple incoming edges and multiple outgoing edges to represent multiple causes and multiple outcomes, respectively. G can provide all the information needed by the diagnosis system. If one packet experiences higher accumulative delay than a threshold, the diagnosis system will be triggered to start constructing the provenance graph of this high latency problem. A typical provenance graph query would require tracing back all the events which receive and send packet p in the graph G and also those send events to the same port P as the packet p at each hop.

In order to capture the provenance of events, a diagnosis system needs to perform two tasks. First, the monitoring component detects the anomalies—i.e., high latency. Second, the debugging component needs to be invoked to nd the root cause of anomalies based on the telemetry information provided by all the related nodes in the problems' provenance graphs. Besides, minimizing the volume of the retrieved telemetry data is also important for scalability.

Precision.The precision requirement refers to the need to capture all relevant events and their timing. Performance degradation problems happen sporadically in the network, so the accumulated delay D at each hop of every ow needs to be monitored to capture all problems in the network. Ideally, we only collect information from all the relevant nodes in the problem's provenance graph [8].

Scalability. To make the monitoring and debugging system scalable, we need to keep the overhead of the whole system low. A related consideration, for instance, is how much involvement of a controller is required for problem diagnosis. The debugging information sent to the analyzer should also be tailored to reduce data volume.

2.3 Existing Solutions Fall Short

Existing solutions all fall short in simultaneously meeting both precision and scalability requirements.

Monitoring solutions. Network monitoring systems install monitoring agents in switches or hosts. For the monitoring system implemented in the switches, normally the agents will report the data extracted from the ows to a central controller for analysis, like LOCO [1], Netow [2], sow [4] and owradar [20]. These solutions can detect the problem and perform some simple diagnosis based on the telemetry data collected from the network. However, the overhead of collecting and analyzing such data is very high, because the telemetry data is collected by the monitoring system constantly. Other solutions like NetSight [14] collect information network-wide, even on network nodes that are not relevant to the problem, making it hard to scale. There are also some solutions combining in-network and end-host monitoring, e.g., SwitchPointer [34] and PathDump [30]. However, since they need to retrieve data from multiple switches and hosts, a central controller must be invoked to retrieve the data from all relevant nodes. Due to the slowness of this process, the relevant information that needs to be sent to the analyzer might have been purged from memory by the switch.

Query-driven solutions. These solutions compile queries into telemetry programs and collect data from all the query-related network nodes. Example systems in this class include Sonata [13] and Marple [25]. They require that the operators know the nature and location of the problems. Performance problems are different because they could arise from random congestion—the problem may happen at random switches sporadically. Query-driven solutions need to monitor all the switches and collect information continuously, which is resource-intensive and unscalable.

2.4 The SpiderMon System

SpiderMon uses packets to carry latency information and detects accumulated latency inside the network, and uses the switch hardware to provide ow-level information and ow contention information to identify the root cause of the problem. As shown in Fig. 2, SpiderMon uses an always-on performance monitor to capture the high accumulated latency problems and other performance degradation events inside the network. The causality data structure keeps track of the most recent contention information. The telemetry data structure preserves the most recent packet-level information in a logically circular buffer.

Figure 2: System architecture

3 DESIGN

3.1 Problem Detection

The accumulated queuing latency is used to identify the ow's congestion level. SpiderMon monitors every ow at every hop by piggybacking the accumulated delay information with an additional header eld L. Whenever a packet enters the egress pipeline, L will be updated by adding the queuing delay $L = L + queue\_time\_delta$. Then it will be compared with the threshold MAX_L to check whether there is an accumulated performance problem. In contrast to leveraging switch data structures to remember latency information, this method does not require the switches to be synchronized. Also, in contrast to storing per-hop latency information in multiple headers, the accumulated latency eld guarantees that one header is enough regardless of the hop count. Once the problem has been detected, the latency monitor will notify the audit request agent in the switch data plane with the 5-tuple of the congested ow, along with its egress and ingress port information. SpiderMon limits the number of events that can be triggered in a period on the same switch. If the queuing delay exceeds a threshold, a global audit request will be broadcasted so information needed for diagnosis is collected once globally, and subsequently all switches are prevented from generating new audit requests for a xed amount of time.

Other than the high accumulated latency trigger, SpiderMon can also support other user-dened triggers such as packet drops, packet timeout events, and pause frames. Take the pause frame as an example: receiving the PAUSE request packet would be a proper trigger for this problem, and once the problem is detected in the switch, SpiderMon can use the same mechanism to trigger the diagnosis procedure for further analysis.

3.2 Provenance Graph Approximation

The audit request agent sends the audit requests from the problematic switch to all relevant switches with a unique event ID. It aims to cover an approximate graph which contains the switches in the provenance graph at low overhead.

Figure 3: Audit requests propagation

We dene two kinds of switches: switches along the historical path of the victim ow are "trunk" switches, and switches which send a large amount of trafc to the "trunk" switches during congestion are "branch" switches. The coverage for a specic problem is a tree whose root is the problematic switch, trunk is the historical path, and the branches are the traversed paths of the interfering trafc. To guarantee the full coverage of relevant switches, the audit requests will also be sent several hops away from "trunk" switches. With a higher hop count, more telemetry information will be collected, and this will also result in higher overhead. Take Fig. 3 as an example: the high latency was detected at switch 7. Then the audit request will be sent to the reverse path of the victim ow—trunk switches 6, 5, and 4, as well as branch switches 0 and 10. SpiderMon uses this as an approximation of the provenance graph. Thus, audit requests are generated at the problematic switch, then propagated back via the victim's historical path and multicasted along the branches at every hop along the reverse path.

SpiderMon chooses to maintain monitoring data that provides provenance in the switch rather than piggybacking it in the packet headers. This is because the packet header cannot carry historical contention information in the network; and 2) the overhead of additional headers increases with the hop count. In contrast, the overhead of maintaining data inside the switch remains the same regardless of the average hop count.

3.3 Supporting Data Structures

SpiderMon introduces causality data structures to help the audit request agent to nd all relevant switches in the provenance graph.

Historical Path Information. SpiderMon uses timeout bloomlter to track the victim ow's historical path. Regular bloomlters allow the insertion of ow IDs and the testing of the presence of a ow ID. However, bloomlter is an accumulative data structure which can only support insertions; its false positive rate will increase with the number of ow IDs inserted. So we need a timeout feature to remove the outdated data from the bloomlter, which requires more memory but provides a "sliding window" of the historical ow information.

For a switch with N ports, each egress pipeline maintains a bloom lter with M rows and N cells per row, and each column represents a bloom lter for the corresponding port. The timeout bloom lter replaces the bit record with a short timestamp, which can be used to remove outdated record when querying the bloom lter. The details about maintaining and querying the bloom lter are shown in

Algorithm 1: Timeout bloomlter data structure

```
Input: B: Timeout Bloomlter, inPort: Incoming port index,
       5 tuple: 5-tuple, TS: Timestamp, isAR: Is audit
       request?
if isAR == False then
    hashValues = HASH 5 tuple
    for hashValue 2 hashValues do
        B hashValue inPort   TS
    end
else
    hashValues  HASH 5  tuple
    ifHit   1
    for hashValue in hashValues do
        ifHit   isValid TS ^ ifHit
    end
    in Return ifHit
end
```

Algorithm 13, Fig. 4(a) and Fig. 4(b). The memory footprint of this data structure can be reduced by shrinking the timeout threshold for the bloom lter, namely, storing the path information for a shorter time. Therefore, there is a trade-off between the length of path history and memory usage.

Flow Contention Information. A per-port per-epoch data structure is used to collect the ow contention information, tracking all relevant ingress ports that are sending trafc to the victim's egress queue. For each egress port, the switch maintains a bit array whose size is the same as the number of switch ports. And each bit in the bit array represents whether a port has sent data to this egress port in the last epoch. All ports with bit 1 will be considered as suspect of contending ows, and the audit request will be sent to them if the audit request of the victim ow is received.

Multicast Group Vector. The per-port per-epoch bitarray serves as the multicast group vector for the audit request multicast. Because switches have limited number of pre-dened multicast groups, the multicast group indexes cannot be mapped to multicast groups arbitrarily. Thus, SpiderMon performs a broadcast and uses the multicast group vector to drop the packets that are not required to be sent from some egress ports. For instance, the vector 0101 will drop packets for port 0 and 2.

3.4 Telemetry Information

SpiderMon requires switches to maintain per-ow records for further analysis. Our main focus here is not to develop a new data structure for monitoring; rather, we explore how SpiderMon can be integrated with existing in-network telemetry systems (e.g., Marple [25], *Flow [28]) and end-host based telemetry systems (e.g., Switch-Pointer [31], Conuo [19]). Below, we analyze SpiderMon's requirement on the time duration for which a switch must maintain telemetry data. Consider the maximum allowed end-to-end delay to be T. The time it takes to propagate audit requests from the initiator to relevant switches—assuming there is no congestion in the reverse path—is half RTT in the worst case. Since the congestion is detected after accumulated queuing delay exceeds the maximum allowed latency, i.e., T, the lower bound on the time duration is T s RTT. For