Software-defined Latency Monitoring in Data Center Networks

Curtis Yu[1], Cristian Lumezanu[2], Abhishek Sharma[2], Qiang Xu[2], Guofei Jiang[2], Harsha V. Madhyastha[3]

[1] University of California, Riverside
[2] NEC Labs America
[3] University of Michigan

Abstract. Data center network operators have to continually monitor path latency to quickly detect and re-route traffic away from high-delay path segments. Existing latency monitoring techniques in data centers rely on either 1) actively sending probes from end-hosts, which is restricted in some cases and can only measure end-to-end latencies, or 2) passively capturing and aggregating traffic on network devices, which requires hardware modifications.

In this work, we explore another opportunity for network path latency monitoring, enabled by software-defined networking. We propose SLAM, a latency monitoring framework that dynamically sends specific probe packets to trigger control messages from the first and last switches of a path to a centralized controller. SLAM then estimates the latency distribution along a path based on the arrival timestamps of the control messages at the controller. Our experiments show that the latency distributions estimated by SLAM are sufficiently accurate to enable the detection of latency spikes and the selection of low-latency paths in a data center.

1 Introduction

Many data center applications such as search, e-commerce, and banking are latency-sensitive [3, 7]. These applications often have several distributed components (*e.g.*, front-end, application server, storage) that need to communicate across low-latency network paths to reduce application response times. To effectively manage data center networks and provide fast paths, operators must continually monitor the latency on all paths that the traffic of an application could traverse and quickly route packets away from high-delay segments [1, 5].

Operators can monitor path latency *from the edge* by sending probes (*e.g.*, ICMP requests) between servers and measuring response times. However, three factors complicate this approach. First, some data centers (e.g., collocation centers [14]) restrict access to customer servers. Second, end-to-end probes cannot monitor the latency on path segments between arbitrary network devices, which is helpful in identifying sources of high delay. Finally, operators are reluctant to repeatedly run expensive measurements from the edge and prefer to allocate server resources to customer VMs [12].

The alternative is to monitor latencies *from inside the network* by capturing information about paths directly from network devices. Trajectory sampling [6] and *l2ping* are examples of this approach. However, all such solutions incur the overhead of performing real-time local coordination and aggregating measurements captured at many devices. Recent work proposes to instrument switches with a hash-based primitive that records packet timestamps and measures network latency with microsecond-level accuracy [10, 11]. However, these methods need hardware modications that may not be available in regular switches anytime soon.

In this paper, we explore another opportunity to monitor path latency in data center networks, enabled by software-defined networks (SDNs). We develop SLAM, a framework for **S**oftware-defined **LA**tency **M**onitoring between any two network switches, that does not require specialized hardware or access to endhosts. SLAM uses the SDN control plane to manage and customize probe packets and trigger notifications upon their arrival at switches. It measures latency based on the the notifications' arrival times at the control plane.

SLAM is deployed on the network controller and computes latency estimates on a path in three steps. *(setup)* It installs specific monitoring rules on all switches on the path; these rules instruct every switch to forward the matched packets to the next switch on the path; the first and last switches also generate notifications (*e.g.*, PacketIn) to the controller. *(probe)* SLAM sends probes that are constructed to match only the monitoring rules and that traverse only the monitored path. *(estimate)* It estimates the path's latency based on the times at which the notification messages (triggered by the same probe) from the first and last switches of the path are received at the controller.

SLAM offers several advantages over existing latency monitoring techniques. First, by exploiting control packets inherent to SDN, SLAM requires neither switch hardware modifications nor access to endhosts. Second, SLAM enables the measurement of latency between arbitrary OpenFlow-enabled switches. Finally, by computing latency estimates at the controller, SLAM leverages the visibility offered by SDNs without needing complex scheduling of measurements on switches or end-hosts. Moreover, SLAM's concentration of latency monitoring logic at the controller is well-suited to the centralized computation of low latency routes that is typical to SDNs.

We address three key issues in our design of SLAM. First, latencies on data center network paths are small—on the order of milli- or even micro-seconds—and vary continually, due predominatly to changing queue sizes. As a result, any single latency estimate may become invalid between when it is measured by SLAM and when it is used to make rerouting decisions. Therefore, instead of a single latency estimate for a path, we design SLAM to infer the latency distribution over an interval. A latency distribution that shows high latencies for a sustained period of time can be more instrumental in inferring high-delay segments in the network.

Second, since SLAM's latency estimation is based on the timings of PacketIn's received at the controller, the accuracy of latency estimates depends on both end switches on the path taking the same amount of time to process notification messages and send them to the controller. However, in reality, the delay incurred in a switch's processing of the action field of a matched rule and its subsequent generation of a notification (*i.e.*, PacketIn) depends on the utilization of the switch CPU, which varies continually. Moreover, switches are generally not equidistant from the controller. To account for these factors, for every switch, SLAM continually monitors the switch's internal control path latency and its latency to the controller (via EchoRequest messages) and adjusts its estimation of the latency distribution.

Lastly, despite SLAM's benefits, its probing overhead is the same as that associated with probes issued from end-hosts. To alleviate this cost, we also explore the feasibility of SLAM in a reactive OpenFlow deployment, where new flows always trigger PacketIn messages from every switch. The key idea is for SLAM to use the existing OpenFlow control traffic without requiring monitoring probes to trigger additional PacketIn messages. We use a real enterprise network trace to show that SLAM would be able to capture latency samples from most switch-to-switch links every two seconds by relying solely on PacketIn's triggered by normal data traffic.

We deploy and evaluate a preliminary version of SLAM on an OpenFlow-based SDN testbed and find that it can accurately detect latency inflations of tens of milliseconds. SLAM works even in the presence of increase control traffic, showing a median latency variation of a few milliseconds when the switch has to process up to 150 control messages per second. Although not suitable to detect very fine variations in latency, SLAM is quick and accurate in identifying high-delay paths from a centralized location and with little overhead.

2 Background

We first describe the operation of a typical OpenFlow network and discuss the factors that contribute to the latency experienced by a packet that traverses it.

2.1 OpenFlow

We consider a network of OpenFlow-enabled switches, connected with a logically centralized controller using a secure, lossless TCP connection. The controller enforces network policies by translating them into low-level configurations and inserting them into the switch flow tables using the OpenFlow protocol.

The network configuration consists of the forwarding rules installed on switches. Every rule consists of a bit string (with 0, 1, and $*$ as characters) that specifies which packets match the rule, one or more actions to be performed by the switch on matched packets, and a set of counters which collect statistics about matched traffic. Possible actions include "forward to physical port", "forward to controller", "drop", etc.

The controller installs rules either proactively, *i.e.*, at the request of the application or the operator, or reactively, *i.e.*, triggered by a PacketIn message from a switch as follows. When the first packet of a new flow arrives, the switch looks for a matching rule in the flow table and performs the associated action. If there

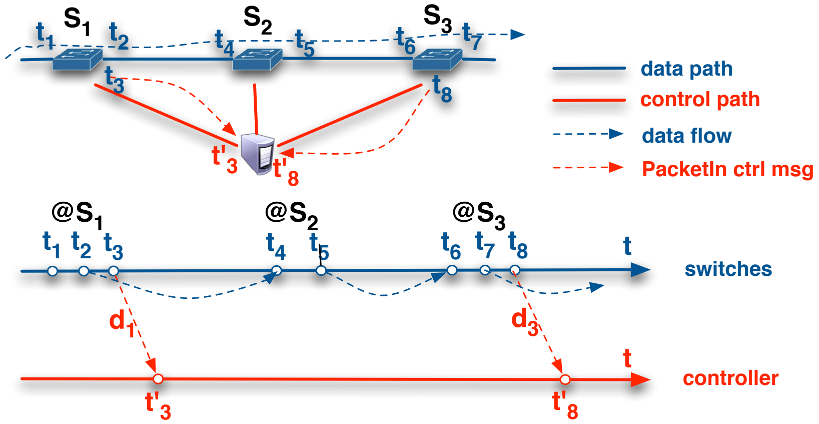


Fig. 1: Latency computation using control message timestamps. Consider a packet traversing a path comprising switches S_1, S_2, and S_3. The packet arrives at these switches at t_1, t_4, and t_6 and leaves at t_2, t_5, and t_7. The true latency between S_1 and S_3 is $t_7 - t_2$. The matching rule at switches S_1 and S_3 has the additional action "send to controller" to generate PacketIn's (the red dotted lines). t_3 and t_8 are the times when the PacketIn's leave S_1 and S_3, and they arrive at the controller at t'_3 and t'_8. d_1 and d_3 are the propagation delays from switches S_1 and S_3 to the controller. We use $(t'_8 - d_3) - (t'_3 - d_1)$ to estimate the latency between S_1 and S_3, after accounting for the processing times in each switch (see Section 3).

is no matching entry, the switch buffers the packet and notifies the controller by sending a PacketIn control message containing the packet header. The controller responds with a FlowMod message that installs a new rule matching the flow into the switch's flow table. The controller may also forward the packet without installing a rule using a PacketOut message.

2.2 Data center path latency

A packet traversing a network path experiences *propagation delay* and *switching delay*. Propagation delay is the time the packet spends on the medium between switches and depends on the physical properties of the medium. The propagation speed is considered to be about two thirds of the speed of light in vacuum [16]. The switching delay is the time the packet spends within a switch and depends on the various functions applied to the packet. In general, the switching delay in an OpenFlow switch has three components: lookup, forwarding, and control. We describe them below and use Figure 1 to illustrate.

Lookup. When a switch receives a packet on one of its input ports, the switch looks for a match in its forwarding table to determine where to forward the packet. This function is usually performed by a dedicated ASIC on parts of the packet header.

Forwarding. A matched packet is transferred through the internal switching system from the input port to an output port. If the output link is transmitting another packet, the new packet is placed in the output queue. The time a packet spends in the queue depends on what other traffic traverses the same output port and the priority of that traffic. In general, forwarding delays dominate lookup delays [16]. The intervals $[t_1, t_2]$, $[t_4, t_5]$, and $[t_6, t_7]$ represent the combined lookup and forwarding delays at switches S_1, S_2, and S_3 in Figure 1.

Control. If there is no match for the packet in the flow table or if the match action is *"send to controller"*, the switch CPU encapsulates part or all of the packet in a PacketIn control message and sends it to the controller. The control delay is the time it takes the PacketIn to reach the controller ($[t_2, t'_3]$ and $[t_7, t'_8]$ in Figure 1).

3 Latency monitoring with SLAM

SLAM computes the latency distribution for any switch-to-switch path by gathering latency samples over a specified period of time. We define the latency between two switches as *the time it takes a packet to travel from the output interface of the first switch to the output interface of the second switch*, *e.g.*, the latency of the path (S_1, S_3) in Figure 1 is $t_7 - t_2$. Our definition of latency does not include the internal processing of the first switch, $t_2 - t_1$, on the path due to the way we use OpenFlow control messages as measurement checkpoints. However, since we continually monitor internal processing delays (see later in the section), we can account for any effects they may have on the overall latency estimation.

Directly measuring the time at which a switch transmits a packet is either expensive [6] or requires modifications to the switch hardware [10]. Instead, we propose that switches send a PacketIn message to the controller whenever a specific type of data packet traverses them. We estimate the latency between two switches as the difference between the arrival times at the controller of PacketIn's corresponding to the same data packet, after accounting for the differences in internal processing of the two switches and propagation delays to the controller. In Figure 1, the estimated latency is $(t'_8 - d_3) - (t'_3 - d_1)$.

We incorporate these ideas into the design of SLAM, an OpenFlow controller module that estimates the latency distribution between any two OpenFlow switches in a network. Next, we discuss how to generate and send probes that trigger PacketIn messages and how to calibrate our latency distribution to the differences in control processing latency between switches. We then describe the design of SLAM.

3.1 Latency monitoring

To estimate latency on a path, SLAM generates probe packets that traverse the path and trigger PacketIn messages at the first and last switches on the path. To guide a probe along an arbitrary path, we pre-install forwarding rules at switches along the path, whose action field instructs the switch to send matched packets to the next-hop switch. In addition, to generate PacketIn's, the rules at the first and last switch on the path contain "send to controller" as part of their action set. SLAM sends monitoring probes using PacketOut messages to the first switch on the path. Our method is similar to the one proposed by OpenNetMon [15], but we explore the implications of using such a system, including its issues, and quantify this effect on the final result.

An important requirement is that the monitoring rules we install to guide the probes do not interfere with normal traffic, *i.e.*, only our probes match against them. For this, we make the rules very specific by not using wildcards and specifying exact values for as many match fields as possible (*e.g.*, VLAN tag, TCP or UDP port numbers, etc.). To save space on switches, we also set the rules to expire once the monitoring is finished by setting their hard timeout.

3.2 Control processing

We define the *control processing time* of a switch as the time it takes a switch to process the action included in the rule that matches a packet, generate a PacketIn, and send it to the controller. In Figure 1, $t'_3 - t_2$ and $t'_8 - t_7$ are the control processing times for S_1 and S_3. Control processing times determine when PacketIn messages arrive at the controller. If processing times vary across the first and last switch, the latency estimation on the path is skewed.

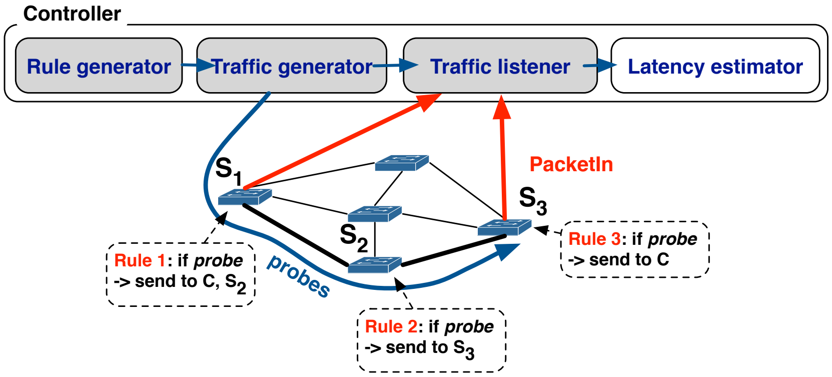
Control processing consists of slow path delay and control channel delay. The slow path delay is the time it takes the switch to transfer the packet along its internal circuits from the ASIC where the match is performed to the switch CPU that generates the PacketIn. As shown in prior work [4], the slow path delay depends on what other operations (*e.g.*, flow installations, stat queries) are performed simultaneously on the switch. The control channel delay is the propagation delay from the switch to the controller.

We adapt to the variations in control processing across switches by constantly monitoring both the slow path and control channel delays. To monitor the slow path delay of a switch, we send packet probes to the switch using PacketOut, use a carefully placed rule to trigger a PacketIn, and then drop the probe without forwarding it to other switches. This resembles our path latency estimation method described above, with the modification that the path to be monitored consists of one switch. We discard latency samples obtained during periods when the slow path delays of the first and last switches on a path vary. Predicting how each variation affects our latency estimate is subject of future work.

To monitor the control channel delay on a switch, we send EchoRequest OpenFlow control messages to the switch and measure the delay in its replies. We find that the control channel delay from the controller to switch is more predictable. Thus, if we discover that switches are not equidistant to the controller, we simply adjust the estimated latency by the difference in their control channel delays, as hinted earlier in the section.

3.3 Monitoring design

We have developed SLAM, a framework for latency monitoring in SDNs, based on the methods enumerated above. SLAM combines four components—rule gen-



Controller
Rule generator
Traffic generator
Traffic listener
Latency estimator
PacketIn
S_1
S_2
S_3
probes
Rule 1: if *probe* -> send to C, S_2
Rule 2: if *probe* -> send to S_3
Rule 3: if *probe* -> send to C

Fig. 2: SLAM design. SLAM generates probe packets along the path to be monitored. The probes are guided by carefully specified rules and trigger PacketIn messages at the first and last switches on the path. SLAM analyzes PacketIn arrival times and estimates path latency.

erator, traffic generator, traffic listener, and latency estimator—that run on the network controller (Figure 2).

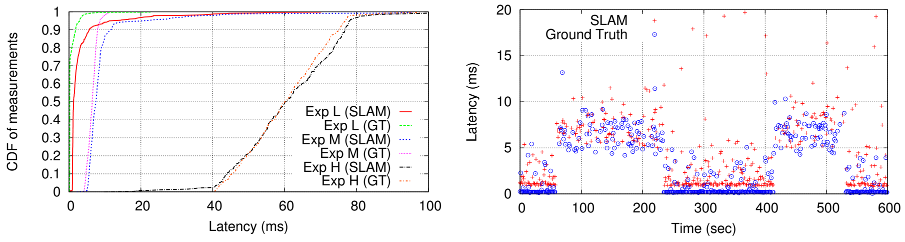
Given a path to monitor, SLAM identifies the first and last switches on the path. It then installs a specific rule on each switch on the path to guide measurement probes, as explained above. The traffic generator then sends a stream of packet probes along the monitored path using OpenFlow PacketOut messages. These packets match the specific rules installed in the previous step. Normal traffic is processed by the original rules on the switches and is not affected by our monitoring rules. In addition, the measurement module generates probes to monitor the slow path and control channel delays of the first and last switches on a monitored path.

The traffic listener captures control packets received from switches and records their arrival timestamps. To obtain a latency sample, it then correlates PacketIn messages associated with the same probe packet and triggered by different switches. By aggregating the latency samples obtained from multiple probes sent on a path, SLAM computes a latency distribution for the path.

4 Evaluation

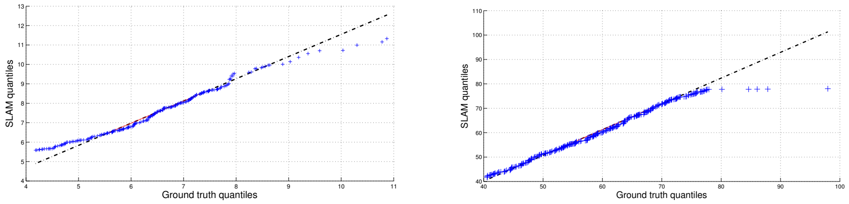
We implemented SLAM as a module for the POX OpenFlow controller and deployed it on our 12-switch network testbed. We evaluate SLAM from three aspects: (1) the accuracy of its latency estimates, (2) its utility in selecting paths based on latency, and (3) its adaptiveness to network conditions.

Ground truth estimation. To evaluate the quality of SLAM's path latency estimates, we must first measure the real path latency (*i.e.*, the ground truth). As we cannot directly time packet arrival and departure on switches, we use the following setup to measure ground truth, similar to that used for OpenFlow testing by Rotsos *et al.* [13] and by Huang *et al.* [8]. We create another physical connection between the first and last switches on a path and the controllerin



(a) (b)

Fig. 3: (a) SLAM vs. Ground truth latency empirical CDFs. (b) SLAM with bursty traffic. As path latency increases, SLAM is able to correctly detect the increase.



(a) (b)

Fig. 4: Quantile-Quantile plots for SLAM vs. ground truth in (b) *Exp M*, and (c) *Exp H*. The quantiles for SLAM's estimates are close to the quantiles for ground truth estimates, indicating that SLAM is able to detect millisecond-level path latency variations.

addition to the already existing control channel and put the controller on the data plane.

We use the controller to send probe packets along the path to be monitored. When a probe arrives at the first switch, the action of the matching rule sends the packet both to the next switch on the path and to the controller on the data plane. Similarly, at the last switch, the matching rule sends probe packets back to the controller. We obtain the ground truth latency by subtracting the two arrival times of the same probe at the controller. This method is similar to that used by SLAM, with the difference that no packet ever crosses into the control plane. Although the computed latency may not perfectly reflect the ground truth, it does not contain the effects of control processing, and hence, can be used as a reasonable estimate to compare against SLAM's estimated latency distribution.

Experiments. To evaluate SLAM's performance under different network conditions, we perform three sets of experiments: low latency (*Exp L*), medium latency (*Exp M*), and high latency (*Exp H*). We estimate latency between the same pair of switches in our testbed, but each of the three experiments takes place on a different path between the switches. There is no background traffic for the low latency experiment. For medium and high latency experiments, we introduce additional traffic using *iperf* and simulate congestion by shaping traffic at an intermediate switch on the path. We use 200 Mbps *iperf* traffic with 100 Mbps traffic shaping in *Exp M*, and 20 Mbps *iperf* traffic with 10 Mbps traffic shaping in *Exp H*. In each experiment, we run both SLAM and the ground truth estimator concurrently for 10 minutes with a rate of one probe per second.

4.1 Accuracy

First, we seek to understand how similar to ground truth is the latency distribution computed by SLAM. To compare two latency distributions (of different paths or of the same path under different conditions), we use the Kolmogorov-Smirnov (KS) test [9]. The KS test computes a statistic that captures the distance between the empirical cumulative distribution functions (CDFs) of the two sets of latencies. The null hypothesis is that the two sets of latencies are drawn from the same distribution. If we can reject the null hypothesis based on the test statistic, then this implies that the two distributions are not equal. Further, we can compare the quantiles of the two distributions (*e.g.*, median) to determine if one path has lower latencies than the other. Figure 3(a) shows that, although SLAM overestimates the ground truth for under-millisecond latencies, it is able to match the ground truth latency distribution as the path latency increases. Indeed, the KS test does not reject the null hypothesis for *Exp M* and *Exp H*.

Figures 4(a) and 4(b) show the Quantile-Quantile (Q-Q) plots for *Exp M* and *Exp H*, respectively. We remove outliers by discarding the bottom and top 10% (5%) of SLAM's latency estimates for *Exp M* (*Exp H*). Except for a small number of very low and high quantiles, the quantiles for SLAM's estimates are equal or close to the quantiles for ground truth estimates; most of the points in the Q-Q plot lie on the $y = x$ line.

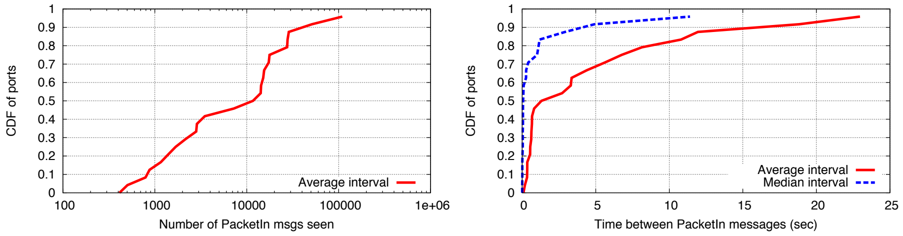
4.2 Filtering out high latency paths

SLAM can help network operators identify low-latency paths. For a collection of paths, we can use the pairwise KS test to first select a subset of paths whose distribution are different from each other, and then filter out paths with high latency quantiles. Similarly, when monitoring a path, an operator can first use the KS test to determine if its latency distribution has changed (*e.g.*, due to change in traffic) and then use the latency quantile values to decide whether to continue using it or switch to a different path. For instance, in our experiments, when we compare samples from *Exp M* and *Exp H*, the KS test rejects the null hypothesis, *i.e.*, the latency distribution on the monitored path has changed due to change in traffic. Table 1 shows that four quantiles for the two samples differ significantly. This is confirmed by Figure 3(a), where empirical CDFs of the measurements collected by SLAM for *Exp M* and *Exp H* are clearly different. SLAM's use of KS test, in combination with latency quantiles, is more robust because an operator can be confident that the difference in latency quantiles across paths or on the same path over time is statistically significant.

4.3 Sensitivity to network conditions

Next, we study SLAM's accuracy in the presence of bursty data traffic and increased control channel traffic.

Data traffic. To see if variable traffic affects SLAM's latency estimates, we repeat *Exp H*, but instead of running *iperf* continuously, we run it in bursts of



(a) (b)

Fig. 5: (a) No. of PacketIn's each link in a 24 port switch sees in three hours. (b) Average and median time between PacketIn's per link on a 24 port switch.

Exp #	50th %tile	75th %tile	90th %tile	95th %tile
Exp M	7.47 ms	8.66 ms	11.6 ms	19.2 ms
Exp H	60.0 ms	71.9 ms	76.8 ms	78.0 ms

Table 1: Comparison of the 50th, 75th, 90th, and 95th percentile values for *Exp M* and *Exp H*.

variable size. Figure 3(b) shows how latency varies over time as we introduce and remove traffic from the network. SLAM's estimates adapt well to changes in the ground truth latency triggered by introducing congestion in the network. Like the results shown in Figure 3(a), SLAM over-estimates latency when path latency is low but accurately captures latency spikes. These results further confirm SLAM's effectiveness in enabling data center networks to route traffic away from segments on which latency increases by tens of milliseconds.

Control traffic. We monitor the slow path delay of switches in our network while we introduce two types of control traffic: FlowMod, by repeatedly inserting forwarding rules, and PacketIn, by increasing the number of probes that match a rule whose action is "send to controller". We varied the control packet rate from 1 to 20 per second and observed a median increase of 1.28 ms. Varying the amount of concurrent rule installations from 0 to 150 rules per second resulted in a median increase of 2.13 ms. Thus, the amount of unrelated control traffic in the network does not influence SLAM's effectiveness in detecting high-delay paths.

5 Reactive OpenFlow deployments

So far, we considered a proactive OpenFlow deployment for SLAM, where normal data packets always have a matching rule and do not trigger PacketIn messages. Another option is to use a reactive deployment, in which switches notify the controller of incoming packets without a matching rule by sending a PacketIn control message. Because too many such control messages could overload the controller and make the network unusable [2], reactive deployments are limited to smaller enterprises and data centers with tens of switches or when the network must react to traffic changes automatically.

Reactive networks provide a significant advantage for SLAM: it can use existing PacketIn messages to compute path latency distributions. This eliminates the need to insert expensive probes to trigger PacketIn's and reduces the cost of monitoring by using already existing control traffic [17]. However, there are two disadvantages, which we discuss at large next.

5.1 Variations in control processing

Using reactive PacketIn's at both ends of a path to capture its latency means that normal data packets are delayed at the first switch until the controller tells the switch what to do with them. This introduces an additional delay in the path of a packet described in Figure 1: the time it takes the controller to process the packet and reply to the switch (either with FlowMod or PacketOut) and the time it takes the switch to forward the packet to the out port once it learns what to do with it. SLAM can estimate the controller processing time and the controller-to-switch delay as described in Section 3.2. However, the switch forwarding time depends on the load on the switch CPU and what other traffic is traversing the switch; this is more difficult to estimate accurately. In practice, SLAM can use the approach in Section 3.2 to infer variations in switch processing and discard measurements performed during times when variations are high.

5.2 Frequency of control traffic

The accuracy of SLAM's estimated latency distribution depends on the frequency of PacketIn's from switches at both ends of the measured path. This is affected by the overall distribution of traffic in the network and by the structure of rules used to guide the traffic. For example, because switches on a backup link see little data traffic, they trigger little control traffic for SLAM to use. Similarly, forwarding rules with long timeouts or with wildcards limit the number of PacketIn messages.

To evaluate the frequency of PacketIn measurements, we simulate SLAM on a real-world enterprise trace. We use the *EDU1* trace collected by Benson *et al.* [2], capturing all traffic traversing a switch in a campus network for a period of three hours. We identify all network flows in the trace, along with their start time. The collectors of the trace report that the flow arrival rate at the switch is on the order of a few milliseconds [2].

Since only PacketIn's associated with traffic that traverses the same path are useful, we need to evaluate the flow arrival rate for each input port of the switch. Our traffic trace does not contain input port information, therefore we simulate a 24-port switch using the following heuristic. We first associate every distinct $/p$ prefix (where p is, in turn, 32, 30, 28, 20, or 24) of source IP addresses in the trace with a port and then assign each individual flow to the link (or input port) associated with its source IP $/p$ prefix. We group flows by prefix because routing in the Internet is typically prefix-based. Below, we present results for $p = 28$; results for other prefix lengths are qualitatively similar.

We compute both the number and the frequency of PacketIn messages that each link receives during the measurement period. Figure 5(a) shows that most links see more than 10,000 PacketIn's during the three hour span, which is equivalent to a rate of around one PacketIn per second. Figure 5(b) presents the average and median time between consecutive PacketIn's for each link of the switch. SLAM would capture samples from most links every two seconds and 80% of all links would be measured less than every 10 seconds.

To summarize, our analysis on a real-world enterprise trace shows that, in a reactive SDN deployment, SLAM would be able to capture latency measurements once every two seconds on average without requiring any additional generation of probes. We are currently investigating the design of an adaptable SLAM that would rely on existing PacketIn's when control traffic volume is high and generate probes that trigger artificial PacketIn's when control traffic is scarce.

6 Conclusion

We presented SLAM, a path latency monitoring framework for software-defined data centers. SLAM uses timestamps of carefully triggered control messages to monitor network latency between any two arbitrary switches and identify high-delay paths. SLAM's measurements are accurate enough to detect latency inflations of tens of milliseconds and enable applications to route traffic away from high-delay path segments.

References

1. M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *USENIX NSDI*, 2010.
2. T. Benson, A. Akella, and D. Maltz. Network traffic characteristics of data centers in the wild. In *ACM IMC*, 2010.
3. Y. Chen, R. Mahajan, B. Sridharan, and Z.-L. Zhang. A provider-side view of web search response time. In *Proc. ACM SIGCOMM*, 2013.
4. A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *Proc. ACM SIGCOMM*, 2011.
5. A. Das, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and C. Yu. Transparent and efficient network management for big data processing in the cloud. In *HotCloud*, 2013.
6. N. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. In *Proc. ACM SIGCOMM*, 2000.
7. T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheong, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing web latency: The virtue of gentle aggression. In *Proc. ACM SIGCOMM*, 2013.
8. D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity switch models for software-defined network emulation. In *Proc. HotSDN*, 2013.
9. A. N. Kolmogorov. Sulla determinazione empirica di una legge di distribuzione. *Giornale dellIstituto Italiano degli Attuari*, 4(1):83–91, 1933.
10. R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese. Every microsecond counts: Tracking fine-grain latencies with a lossy difference aggregator. In *Proc. ACM SIGCOMM*, 2009.
11. M. Lee, N. Duffield, and R. R. Kompella. Not all microseconds are equal: Fine-grained per-flow measurements with reference latency interpolation. In *Proc. ACM SIGCOMM*, 2010.
12. M. Moshref, M. Yu, A. Sharma, and R. Govindan. Scalable rule management for data centers. In *Proc. USENIX NSDI*, 2013.
13. C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. OFLOPS: An open framework for OpenFlow switch evaluation. In *Proc. PAM*, 2012.
14. RagingWire. `http://www.ragingwire.com`.
15. N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers. OpenNetMon: Network Monitoring in OpenFlow Software-Defined Networks. In *IEEE NOMS*, 2014.
16. G. Varghese. *Network Algorithmics*. Elsevier/Morgan Kaufmann, 2005.
17. C. Yu, C. Lumezanu, V. Singh, Y. Zhang, G. Jiang, and H. V. Madhyastha. Monitoring network utilization with zero measurement cost. In *Proc. PAM*, 2013.