Software-defined Latency Detection and Rerouting

Baohe Zhang, Yuying Chen, Kechen Lu

University of Illinois at Urbana-Champaign

ABSTRACT

In today's data center, most platform, infrastructure and services deployed over it have dramatically increasing demands of low-latency networks for latency-sensitive applications, such as high-frequency trading, online gaming and other performance critical distributed systems. End-to-end latency monitoring and measurement in the traditional way are not appropriate and adapted to the new requirements of switch-switch latency detection. With the power of SDN (Software Defined Network) and programmable data plane and control plane separation, we propose a new tool framework, LMPS (Latency Monitor and Path Selector), to address the switch-switch latency detection for various connectivity paths of assigned start switch and end switch, and with the estimated low latency path to perform the simple path rerouting. After evaluation for our approach, with specific workload and latency patterns, we found our approach could detect the low latency and enforce the rerouting to it which successfully lower the latency performed by ping.

1. INTRODUCTION

As the increase of demands for low-latency networks over the deployment of latency-sensitive applications such as distributed systems over e-comm, high-frequency trading, online gaming, and so forth, service providers and operators would want to continuously maintain the traffic balanced intranet and prevent congestion conditions inside the data center networks. To capture more accurate global view of the latency distribution over central network in the data center, start switch to end switch monitoring plays a more important role instead of end-to-end latency detection. For the reduction of overhead over some paths in the network, detection of high-latency paths and rerouting for packet forwarding is the basic way to address it. In the traditional network traffic engineering and monitoring, the approaches to address it are inflexible and hard to accommodate for various network typologies, plus the end-to-end latency measurement is not a feasible way to reflecting the condition of intermediate part of the network paths. So we propose a software defined network approach to perform the latency detection and packets rerouting with low-latency path.

Since software defined network has become a practical and popular approach to address today's network related problems, in the use case and scenarios we described above, programmable control plane and data plane could help address the case we proposed. Firstly, with the help of SDN-enabled latency monitor, we could detect the path latency accurately with only trivial performance impacts. Secondly, measurements over critical paths of the networks could bring less performance impacts to the existing traffic and simplify the rerouting setup. To be brief, the core question we focus is how can we build a SDN-enabled monitor to address more accurate and trivial performance-impact measurement of latency of critical paths in latency-sensitive networks, and avoids the high latency paths.

The end-to-end ping latency measurement like we normally do in endhost or routers is not feasible if without additional hardware like new hosts. Based on the scenarios mentioned above, after separation of the data plane and control plane, the behaviors of switches could be more controllable, and thus implementing a latency detector and path selector becomes easier and valuable in the data center networks with low-latency requirements. In this paper, we propose a new framework, LMPS, to perform the Latency Monitoring and Path Selecting, and we have the following contributions:

- An exploration of the SDN control plane driven application, which tries to make more fine-grained and event-based network telemetry and interactive control over network topology.
- Design and implement of an OpenFlow controller, which combines the latency monitoring and path selector, two components utilizes the capabilities of control plane to have accurate measurements but only bringing trivial performance impacts.

2. BACKGROUND AND RELATED WORK

2.1 Background

OpenFlow[7] is a communication protocol enables control over forwarding plane, and therefore is especially suitable for running experimental protocols on transport layer. The control provided by OpenFlow is through programming the flow-table in individual switches. When packets arriving, OpenFlow switches will check if the packet matches the entry in flow-table and accordingly execute actions.

An entry in flow-table is composed of three fields – the header to match with flow, the action that decided how to process the packet, the statistics to keep track of passed-through flow. In general, OpenFlow Switches support three basic actions: (1) forwarding packets to given ports, (2) forward packets to controller through secure channel, (3) drop packets. Controller will manage the installation end removal of flow entry for connected switches, providing flexibility of dynamic changing the forwarding rules. When the packet unable to match with any flow entry at switch, the packet will be delivered to controller and trigger the function PacketIn in controller. This feature is generally used to adding rules for a new flow. In our implementation, we also make use of the feature to acknowledge controller the reception of a probe packet.

2.2 Related Work

Traditional techniques in monitoring latency within datacenter including 1) end to end measurement by actively sending probes from hosts and 2) passively capturing aggregating traffic by network hardware. As interest in software-defined network increases, traffic monitoring methods using SDN are also developed in recent years. The framework SLAM[8] measures latency between switches by triggering control messages with probe package and is sufficiently accurate for path selection. Our project based on this framework for package rerouting.

Joan M. purposes a method to minimizing latency of critical traffic[6] using similar method to get the latency between switches. Then they added up the latency in critical path and try to find the path with minimum latency sum. Our project measures minimum latency in different way. We first calculate all possible paths between two critical points. Then we measure end-to-end latency for all paths, by configuring the route in P4. Measuring latency end-to-end can take the switches' processing time into consideration and reduce error.

Microsoft purposes the system Pingmesh[4] to monitor the status of a large scale data center. There are three components in the system - Pingmesh Agent which runs at every server, Pingmesh Controller to decide hwo servers probe each other and DSA that stores data from Pingmesh Agent and processed in a data analysis pipeline. Controller generates list for every server to Ping by its algorithm. Agent downloads and pings the server on list, upload latency data. This process is done using RESTful WebAPI so that controller avoid pushing data to lots of servers. Latency monitoring in Pingmesh is also done by sending probe packets, but without the participation of controller. So each server measures route-trip-time (RTT). Besides latency between servers, Pingmesh also measures packets drop rate using RTT and TCP timeout for SYN packet. For example, if the initial timeout for SYN packet is 3 seconds, 9 seconds means there are two SYN packets dropped. They proposed an estimation of packet drop rate by:

$$\frac{probes\_with\_3s\_rtt + probes\_with\_9s\_rtt}{total\_successful\_probes} \tag{1}$$

One load balancing mechanism of congestion-aware is CONGA[1]. Rather than doing load balancing decision per packet or per flow, CONGA uses flowlets for decision making to achieve more efficiency and fine-grained control. The source switches in CONGA keep the Flowlet Table to look up route for subsequent packets. Previous decision for a flowlet is cached in the entry of Flowlet Tables and will expire if no packets of that flowlet arrives for a period of time. In CONGA, load balancing decision is made based on both uplink congestion metrics and remote congestion metric to minimize the maximum load. To estimate the load of a link, DRE(Dis-counting Rate Estimator) module is introduced. DRE measures the load by counting packet sent over the link and decremented periodically to estimate packets reaching the end. Congestion metric will be carried in packet and keep updating through transferring to destination. When there is a packet of reverse direction, the metrics can be piggybacked from destination switch and update source switch's congestion metric. This mechanism is simple enough to be implemented in switch hardware and is robust to asymmetric congestion.

Another congestion-aware load balancing mechanism DRILL[2] forwards packets based on the thought of Equal Split Fluid (ESF), equally assigning flow to all the least-cost path to destination. This mechanism assume switches keep records of next-hop candidates for each destination, and have access to the outgoing queue. For every input packet, DRILL choose the output port among d ports randomly chosen from candidates, and m ports with least load cached at previous operation, and route the packet to the best candidate. The time complexity for forwarding a packet therefore is $O(d + m)$. The real-world network could be asymmetry due to its original topology and difference in devices. To handle this situation, DRILL introduces weight and score to describe the capacity and feature of paths, as well as the capacity factor to describe the capacity of links on a path. This optimization solve the drawback of ESF and maximum the throughput. DRILL achieves good performance in load balancing while retains most of the simplicity. It also has the defect of performance degraded when encountering too much memory and too many choices.

3. DESIGN

Consider a datacenter with complicated topology and various flow on each path. Congestion could happen to a link when there is heavy load. To avoid congestion, we monitor the load of each possible path to destination and reroute packets with the candidate of lowest latency. Our design can be divided into two parts: load monitoring and path selection.

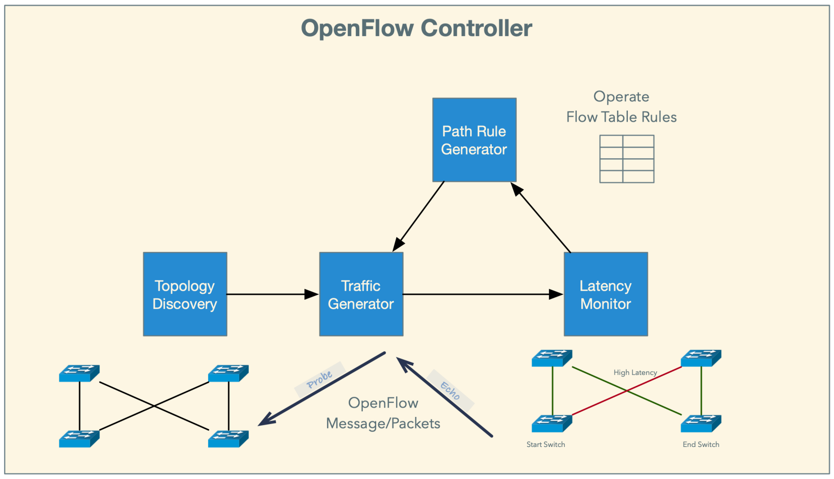

Figure 1: LMPS architecture

We monitor the load of all the paths with specific source and destination. Connection between the source and destination is called Critical Connection and our goal is to minimize latency of this Critical Connection regardless of other background flow.

We display a simple topology in Figure 2. The Critical Connection in this topology is S1 to S2. The candidate paths for the connection is P1: $S1-S3-S2$ and P2: $S1-S4-S2$. We separately detect the load of each path by measurement of their latency.

Latency measurement requires synchronization of clock between device or a reference. Our method measures latency time through the reference of OpenFlow controller, start from controller sending message to source switch, transferring packet through the path, end with a message from destination switch when probe packet arrives.

To improve accuracy, we also measure the route-trip-time (RTT) from controller to switch and subtract RTT from the total latency. We will provide more details in latency measurement in §4.2.

After measuring all paths' latency for Critical Connection, controller chooses the path with lowest latency and install the rule into flow-table.

4. IMPLEMENTATION

4.1 LMPS OpenFlow Controller

OpenFlow [7] controller plays an important role in the framework we proposed, in charge of the control plane management and modification of the data plane flow match tables. As we mentioned above, the controller performs most of the tasks in terms of the events triggered, such as the switch connection up/down, packets in, various OpenFlow message replies in, and the timer events the user setting up. To effectively achieve the goals of our LMPS, we have several component parts to perform different tasks, which includes the topology discovery, path rule generator, traffic generator and latency monitor. The overall architecture of LMPS is shown in Figure 1 below. Topology discovery is the first step following all the openflow-enabled switches are line-up, and then building the adjacent graph. Traffic generator is timer triggered and primarily responsible for the probe packets sending out. Latency monitor is the main measurement component which try to catch up and estimate the latency for the various paths, while the path rule generator would also follow the result of topology discovery and setup the probe-specific forwarding rules, and finally setup rerouting paths after finding out the lowest-latency path.

4.2 Latency Measurement

In the real network world, latency of the whole network includes two part, one is the path latency, and another is the switch latency [8]. Path latency usually constructs by the path data transferring latency over the lines and sometimes depends on the congestion condition in line. But switch latency might contain matching table lookup latency, packet transferring time from the in_port to out_port and the finally if with controller possible there are latency for the control processing which takes over the communication between switches and the controller.

But in this case for our work, we do not care about the exact the each latency part is. The goal of the latency measurement in this paper is detecting the overall switch-to-switch latency in the intermediate path of the network, so the switch latency except control latency, and path latency are all included. Therefore we assume we use the topology as the Figure 3, in the most naive thoughts, we just record the first time the probe packet arrives, and the final time the packet on the destination, and this period is supposed to be the latency we expect.

4.2.1 *Probe Packets Forwarding*

When we do the traffic engineering in networks, we would like to reroute some packets in the specific path and rules, traditionally through protocols like MPLS. MPLS uses labels to direct the packets routes, which is associated with the predetermined path we want the packets to route. But with programmable control plane and SDN, we could generate unique forwarding rules for different path latency probe packets. In other words, we could generate rules for all paths, that the predefined probe packet type in the layer 2 and destination MAC address would be used in the flow table matching fields, and for each probe packets with different "labels", would be directed to the expected destination with installed forwarding rules.

Forwarding Rule Generating To get measurement for all paths from source to destination switch, we execute path discovery. This function runs periodically and therefore new topology can also be discovered. Path discovery will mark all found paths with number. When installing flow-entry to device, the number of path is contained in the rule as a header field to be matched.

Probe Packet Circling Probe packets are triggered by packet_out messages sending by OpenFlow. Source switch receive this message with path number in header field and content of start timestamp. Thus when packet arrives, switch can deliver packet to different outgoing port according to previous-installed rules. When packet arrives at destination switch, the packet was delivered to controller. Controller extract the start timestamp and total transport time for the packet can be calculated.

4.2.2 Latency Estimation

In the controller side, traffic generator would produce probe packets for different paths between start switch and end switch. For example, as the Figure 2 below shows, we have two paths from S1 to S2, $s1 - s3 - s2$ and $s1 - s4 - s2$, after setting up the forwarding matching rules, probe packets with distinct labels would be directed to different paths we want to measure and detect the latency metrics.

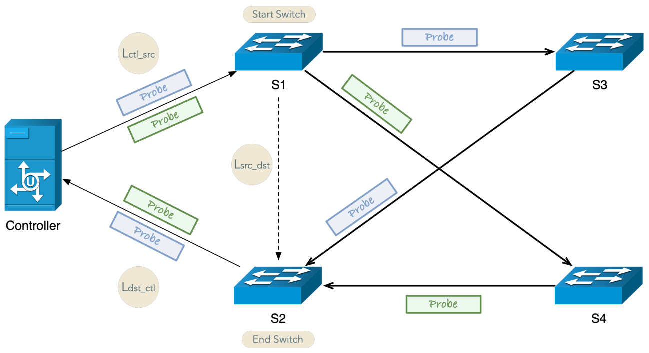

Figure 2: Latency estimation using probe packets for distinct paths

So for one complete path from the probe packets sent to the one received, the latency formulation would be like as below. L_{round} is the duration from controller emitting to receiving this probing message, L_{ctl_src} means the traffic latency from controller to source/start switch, L_{src_dst} is the one from source/start switch to destination/end switch, and L_{dst_ctl} is the latency when probe packets circle back to the controller.

$$L_{round} = L_{ctl\_src} + L_{src\_dst} + L_{dst\_ctl} \tag{2}$$

We can have the measured latency L_{round} as the probe packet sending out timestamp TS_0 and circling back timestamp TS_1 in the controller. The goal of latency measurements is to acquire the accurate L_{src_dst}, so the result could be computed as below. For the estimation of the controller-switch latency L_{ctl_src} *and* L_{dst_ctl}, we would like to play a echo message circling using an OpenFlow message Packet-_Out with assigned output virtual port as controller itself to send to the target OpenFLow-enabled switch, which would be explicitly explained in the following section. For more smoothing result we want to have, there would be a sliding window to get the average latency for distinct paths we could have the traffic from start switch to end switch.

$$L_{src\_dst} = (TS_1 - TS_0) - L_{ctl\_src} - L_{dst\_ctl} \tag{3}$$

Controller-Switch Latency One of the core functionalities of OpenFlow message is to play a role to modify or operate the flow tables of OpenFlow-enabled switches. When measuring controller-switch latency, controller emit a packet_out message to switch, enforcing switch to send a echo-packet back to controller. The probe-packet and echo-packet are distinguished by packet type field. We measure the time of controller from emitting a message to receiving back the echo-packet as the round-trip-time from controller to switch.

Sliding Window After different path latency has been estimated, we would choose the best path with lowest latency, and this path would be leveraged to perform rerouting. To avoid the measurement error brings by bursty traffic, we keep sliding windows with size of 4 for each path's latency measurement and take the average in each sliding window for comparison and lowest-latency path choosing.

4.3 Topology Design

We use Mininet to simulate the network topology. Mininet is a network emulator which creates a network of virtual hosts, switches, controllers, and links. In this topology, we have 4 switches, each denoted as sn. Each switch is connected with a host. Among those 4 switches, there are 4 links, they are $S1-S3$, $S1-S4$, $S2-S3$, $S2-S4$. The Figure 3 below illustrate the topology. The latency of each link can be configured with Linux's tc utility. A pox controller is connected with all switches in this network topology.

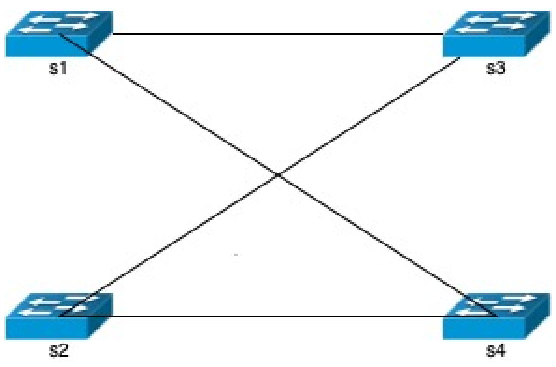

Figure 3: Mininet topology

4.4 Topology Discovery

To calculate the available paths from a source switch to a destination switch, the controller should have the knowledge of the entire network topology. In this project, we use OpenFlow Discovery Protocol (OFDP) to discover the topology. OFDP leverages the Link Layer Discovery Protocol (LLDP) with subtle modifications to perform topology discovery in an OpenFlow network. Pox's openflow.discovery component sends OFDP messages out of OpenFlow-enabled switches to discovery the topology. When a link goes up or down, a LinkEvent will be raised, which contains the information of the newly added or removed link. Among those information, We are interested in source switch's dpid, destination switch's dpid, source port, and destination port. In the project, a component is designed to listen to those events, and builds an adjacency map of the network topology based on these added link information.

A newly adding link will raise two link events, that is because the link is bi-directional. A link from switch1 to switch2 is equivalent to the link from switch2 to switch1. To avoid adding a duplicated link to our adjacency map, we will add the link only when the link events of both directions are raised. When we receive the link event, we will first flip of the link, and check if the link event of the flip link is already captured by the controller. If so, we can add this link to our adjacency map. The adjacency map is a Python dictionary. adjacency[sw1][sw2] is the port from switch1 to switch2, and adjacency[sw2][sw1] is the port from switch2 to switch1.

4.5 Path Finding

To find the best path between two switches, the first step is to find all possible paths between these two switches in the network. Since the controller has the topology information, it can find all possible paths through a backtracking algorithm, that is, we take a path and start walking it. If it leads the destination switch, we store this path and backtrack to another path. If the path doesn't lead us to the destination, vertex, we discard the path. After the program below is terminated, all possible paths from source switch to the destination switch will be populated into the paths variable. Each path contains the information of every hop and the output port of each hop.

```
def get_paths_helper(pre, cur, dst,
                     visited, path, paths):
  visited.add(cur)
  if pre is not None:
    path.append((pre.dpid,
                 adjacency[pre][cur]))

  if cur == None:
    paths.append(path[:])
  elif cur == dst:
    get_paths_helper(cur, None,
                     dst, visited,
                     path, paths)
  else:
    for next_hop, port in
        adjacency[cur].iteritems():
      if next_hop not in visited:
        if port is None:
          continue
        get_paths_helper(cur, next_hop,
                         dst, visited,
                         path, paths)

  if path:
    path.pop()
  visited.remove(cur)
```

4.6 Enforce the Selected Path

Once the best path is found, we should update the flow table of each switch involved in the path to enforce the selected path. For example, we want to set up a packet forwarding path from switch s1 to switch s2, and the best path is [s1, s4, s2]. To Enforce the packets from s1 to s2 all go along this way, we should install flow entries in s1, s4, and s2's flow tables. Since the packets in this s1-s2 path will match on the source mac address and destination mac address, the matching rule will be based on this information. The action is simply forward the packet to next hop via the connected port. More specifically, to instruct the packet in this [s1, s4, s2] path can be correctly forwarded to s4 at s1, we need to install a flow entry at s1, and that is done by sending an OpenFlow Modification message to s1. The flow entry matches on s1's mac address as source mac address and s2's mac address as the destination mac address and has the action that forward this packet to s4 by specifying the output port. The output port from s1 to s4 can be retrieved from our adjacency map. Table 1 shows the flow entries added to s1, s2, s3 during path enforcement.

5. EVALUATION

5.1 Environment Setup

By utilizing the Mininet [5] as the simulated network topology deployment platform, with our POX openflow framework which is the Python implementation of NOX controller

Switch	Match	Action
switch 1	mac src = sender's mac, mac dst = receiver's mac	port = adjacency[s1][s4]
switch 4	mac src = sender's mac, mac dst = receiver's mac	port = adjacency[s4][s2]
switch 2	mac src = sender's mac, mac dst = receiver's mac	port = port connected to receiver

Table 1: Flow entries added to s1, s2, and s3

[3], we would setup a 4-switch topology as the Figure 3 above shows. We use the background traffic and specified link latency in the evaluation experiment to simulate the high latency conditions under the control of the configuration for the topology. Using the VMware to setup the virtual machine of one virtual CPU core i5-4278U @ 2.60 GHz and memory 1024 MB.

5.2 Latency Experiments

We measure the latency for packets sending from switch1 to switch2. We manually add the flow entries to the switches, to make the packet goes through the path s1-s3-s2, and we use Linux's tc utility to configure the link latency of link s1-s3 to 30ms. We compare the result of the latency measured by LMPS and Linux's ping utility. The Figure 4 shows the comparison between the latency estimated by our LMPS and Ping utility of the Mininet commands when there doesn't exist background traffic. The Figure 5 shows the comparison between the latency estimated by our LMPS and Ping utility when there exists background traffic (with the latency of one link set to 30ms).

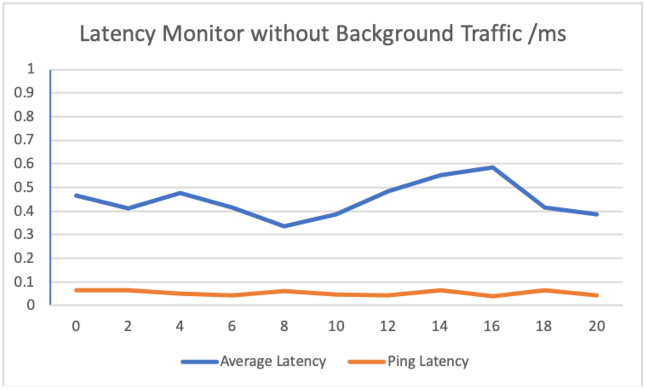

Figure 4: Latency estimated using LMPS/Ping without background traffic latency

5.3 Rerouting Experiment

We measure the latency of h1 ping h2 before and after implementing path selection. There are 2 paths from h1 to h2, one is path1: [h1, s1, s4, s2, h2], another is path2: [h1, s1, s3, s2, h2]. The latency of link s1-s4 is configured as 30ms in the topology. The default path is path1, so the average latency measured by Linux's ping utility before rerouting is 30.485ms. After rerouting, the routing path from h1 to h2 is switched to path2. Hence, the measured average latency is optimized to 0.026ms. The evaluation result for the latency rerouting before or after is shown as below Figure 6.

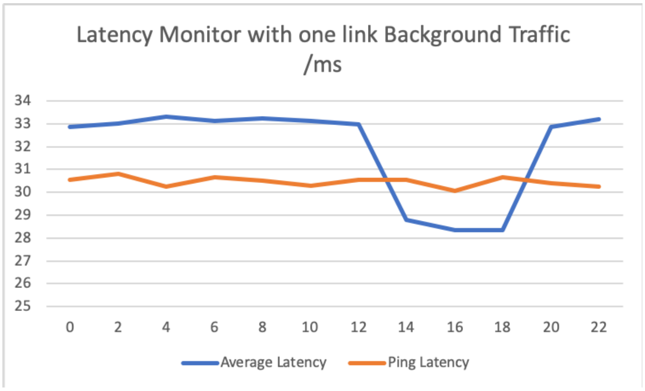

Figure 5: Latency estimated using LMPS/Ping with one link 30ms background traffic latency

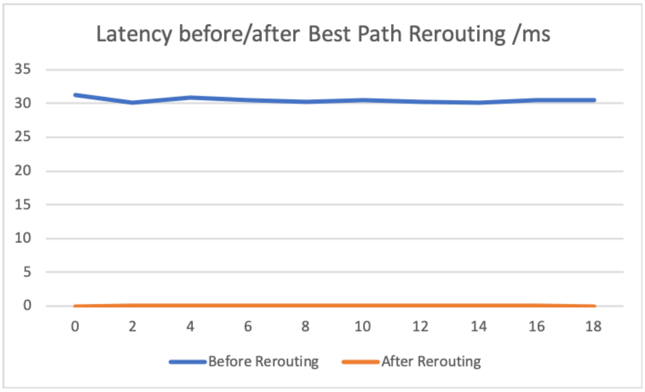

Figure 6: Latency measurement before/after rerouting

6. CONCLUSIONS

The latency measurement and path rerouting tool framework proposed in the paper combines the power of the OpenFlow controller in the control plane, which provides a more flexible and accurate latency measurements approach to address the switch-switch path latency detection. With the utilization of the capabilities in flow table modification, specialized probe packets could be directed to expected path, and path selector and rerouting could be achieved through best low-latency path estimation. As our exploration shows, SDN-powered controller broaden the applications with traditional use cases and provides the data center more flexible capabilities in the latency-sensitive applications and infrastructures.

7. REFERENCES

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

[2] S. Ghorbani, Z. Yang, P. Godfrey, Y. Ganjali, and A. Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 225–238. ACM, 2017.

[3] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[4] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 139–152. ACM, 2015.

[5] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[6] J. M. Llopis, J. Pieczerak, and T. Janaszka. Minimizing latency of critical traffic through sdn. In *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–6. IEEE, 2016.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[8] C. Yu, C. Lumezanu, A. Sharma, Q. Xu, G. Jiang, and H. V. Madhyastha. Software-defined latency monitoring in data center networks. In *International Conference on Passive and Active Network Measurement*, pages 360–372. Springer, 2015.