Chapter 4

Queueing Delay Monitoring in OpenFlow Based SDN Networks

4.1 Introduction

OpenFlow enables the separation of control plane and data plane in Software-defined Networks (SDNs) by providing an interface to programmatically modify or retrieve the switch configuration. As such, it provides a holistic view of the network topology at the controller and enables fine-grained monitoring of the network links. A more precise view of the network provides opportunities for improving the efficiency of routing algorithms.

In recent years, many organisations have been encouraging their employees to use real-time applications such as remote access, voice and video conferencing to make the work environment flexible and increase the work output. This is also fueled by natural disasters such as a pandemic, due to which employees prefer working from home. With the rapid increase in usage of real-time applications, demands for meeting QoS agreements have increased proportionally [143]. IP networks provide best-effort service, i.e., all efforts are made to deliver data but with no guarantee of successful delivery. Real-time applications

- Sandhya Rathee, Shubham Tiwari, K Haribabu, Ashutosh Bhatia. *qMon: A Method to Monitor Queueing Delay in OpenFlow Networks*, Submitted for review 39th IEEE International Performance, Computing and Communications Conference.

need a guarantee of timely arrival of data, which must be provided by the routers and not just the network edges (or the hosts) [144]. Therefore, it is necessary to implement QoS strategies at the network routers to ensure better user experience. In SDNs, this is made possible by taking the QoS decisions at the controller, and enforcing the QoS policies at the switches using OpenFlow [18]. To make efficient routing decisions, the delay should be monitored continuously.

In SDNs, there are various approaches to measure delay [43, 52, 145, 1, 146, 147]. Most of them are active measurement methods, i.e., they construct a specialized control packet (probe packet) and use OpenFlow protocol to inject the probe packet into the data plane. The probe packet physically travels the network links in the datapath and is forwarded back to the controller by a switch at the end of the path. The time taken by the packet to travel the path is used to calculate the delay. This approach suffers from: 1) Data plane footprint, due to injection of probe packets into the data plane 2) Monitoring overhead, due to the processing required at the controller to create probe packets and receive them and 3) Scalability issues, due to increase in the number of probe packets required with the number of switches, links and queues at the egress path. Details about the current approaches for delay measurement and their issues are discussed in Section 4.2.

By studying network traces, authors in [148] have shown that queueing delay can be significant in todays networks. With this motivation, we propose a queueing delay monitoring mechanism (qMon) in OpenFlow based SDNs that leverages a slightly modified OpenFlow message to collect queue statistics. Queue statistics from Open vSwitch [54] are polled at regular intervals and queueing theory is applied at the controller to the aggregated queue statistics to obtain the estimated average waiting time of packets in the queue over a given time interval. Under some assumptions (discussed in Section 4.3), the estimated queueing delay can then be used to further estimate the link latency. This approach addresses most of the issues related to the active delay measurement techniques discussed earlier. The proposed approach relies on the queue statistics message, and no packets are injected into the data plane. Further, qMon can be integrated with existing traffic monitoring modules that poll for queue statistics to estimate traffic load. The proposed approach is scalable, as it requires sending only a single queue statistics request message for estimating link latency at each switch, irrespective of the number of transmit

(TX) ports and queues at the TX ports at each switch. Thus, qMon is scalable with respect to number of switches, TX ports, and queues at the TX ports. A detailed discussion on the issues related to the probe packet based methods, and how qMon addresses these issues are given in Section 4.2.

4.2 Issues and Related Work

Delay measurements can be categorized into active and passive types [149]. Active delay measurement involves constructing a special timestamped control packet (called a probe packet) and injecting it into the datapath. At the end of the path, the receiving node estimates the path delay by calculating the transit time of the probe packet through the path. There are several issues associated with the active delay measurement methods in SDNs:

1. ***Data plane footprint:*** The probe packets injected into the data plane consume the bandwidth of the links involved in the path. The bandwidth consumption increases with an increase in packet injection frequency, leading to a decrease in the available bandwidth and the possibility of change in the traffic behavior [150].

2. ***Monitoring overhead and Scalability:*** Active delay measurement methods require creation of probe packets, injecting them into the data plane, and processing the probe packets at the switches on receiving them. Current active probing methods do not consider the possibility of multiple TC (Traffic Control) queues at the switch's egress ports. For measuring link delay for each of the TC queues, it would be required to send a separate probe packet through each of the queues. With the increase in the number of TC queues at the egress ports and the number of switches, the number of probe packets required increases multiple-folds. This in turn increases the processing overhead on the controller. When measuring end-to-end delay, the number of probe packets increases with an increase in the number of flows. In both the cases, probe packets might be queued at the controller on arrival [151], waiting to be processed. The waiting time of the probe packets at the controller leads to inaccurate link delay measurements.

Passive delay measurements revolve around 1) analytical/statistical methods to model the flow of traffic to estimate delay, or 2) time stamping the existing traffic in the datapath [149]. Limitations of passive methods are inaccuracies in estimating delay. This is due to the assumptions made while employing statistical methods to model the traffic, as it is challenging to have a single model that holds for all traffic distributions. Use of existing traffic in the data plane to measure delay is infeasible in OpenFlow enabled SDNs, as it requires time stamping packets in the data plane and comparing their transit times across two end-points. In OpenFlow it is not possible to tag selective packets with timestamp [1].

Next, we discuss the existing works on measuring the delay in SDN using active and passive measurement methods.

4.2.1 Active Delay Monitoring Methods

Authors in [43, 52, 145, 1, 146, 147] have proposed active delay monitoring schemes in SDNs. The method proposed in OFMon [43] uses OpenFlow to send a time-stamped ethernet frame through a link (say, $S_1 - S_2$, where S_1 and S_2 are switches), and back to the controller as shown in Figure 4.1. The link delay is calculated by subtracting the summation of link delays between controller to switch S_1 and S_2, and a small processing offset introduced by the controller, from the total transit time of the probe packet. The link delay between controller and switches can be estimated by measuring the RTT of statistics request and statistics reply messages. Whereas, the processing offset is calculated for the underlying hardware by measuring the latency on an unused link. Although the size of the probe packet used (24 bytes) is less as compared to Internet Control Message Protocol (ICMP) (196 bytes, request/reply 98 bytes each [43]), the number of packets injected into the data plane increases with the frequency of measurement, leading to an increase in data plane footprint. The method being an active probe method also suffers from active measurement issues such as monitoring overhead and scalability.

In [52, 50, 51], authors use timestamped LLDP packets (which are used for topology discovery by the SDN controllers) to measure the link delay. The method to calculate the link delay is mostly similar to the method described in OFMon [43]. LLDP-looping [145]

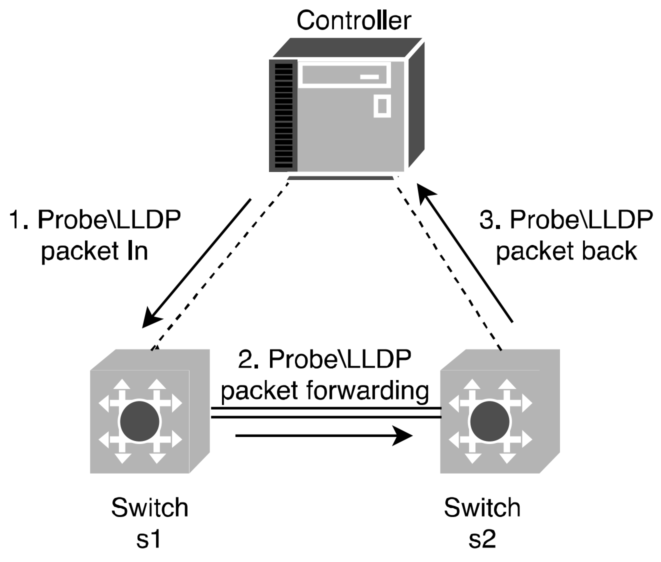


Figure 4.1: Active delay monitoring in SDN networks.

is an extension of LLDP [52], which modifies Open vSwitch to enable looping of LLDP packets between source and destination to measure link RTT with high accuracy. While the methods reuse LLDP packets to reduce the data plane footprint, but since the LLDP packet generation is coupled to the controller's topology discovery module, thus their measurement frequency is limited. With an increase in the packet injection frequency, the method is similar to OFMon [43] and has the same limitations.

OpenNetMon [1] monitors the path delay by injecting a probe packet at the ingress switch and installs the flow entries such that the probe packet travels the monitoring path and the egress switch sends the probe packet back to the controller. Path delay is calculated by subtracting the link delays of the controller to source switch and the controller to destination switch from the difference of probe packet's departure time and arrival time. The controller to switch delay is measured by sending separate probe packets from the controller to both the ingress and the egress switches, which sends the probe packet back to the controller immediately. The works given in [45, 46, 47, 48, 49] use a similar method to measure the path delay. The only difference in [49] is that here every switch mirrors the probe packet and sends it to the controller. Upon receiving the mirrored probe packets, the controller sorts them by their arrival time and estimates the path delay.

GRAMI [146] inserts probe packets from selected nodes in the network to measure

RTT for a single link or between any two switches in the network. It is resource efficient as it does not require involvement of the controller for online RTT monitoring and requires only four flow entries to be installed at every switch. TTL-Looping [147] actively measures end-to-end one-way delay with microsecond precision for flow by multiple iterations of the probe packet along the flow path. This is made possible by storing the required number of iterations in the probe packet's counter field. The flow entries along the path decrease the value of the counter and forward the packet to the next-hop switch. The last switch in the path reverses the direction of the packet. This continues until the counter value becomes zero and the packet matches the flow entry that sends it to the controller. The transit time of the packet is divided by the number of iterations to calculate the RTT. The RTT is then used to compute a one-way delay using the traffic proportion in each direction.

4.2.2 Passive Delay Monitoring Methods

To the best of our knowledge, not much work has been done on passive delay measurements in SDNs. Authors in [53] propose a queueing model for measuring average queueing delay of TCP flows in SDN. The parameters required as input to the method, such as - packet length, buffer size, and link bandwidth, can be computed by querying the switches through OpenFlow. With a growing demand for streaming services, which use UDP as the underlying protocol, the network traffic might contain a high percentage of UDP flows or a heterogeneous mix of multiple flows. Method proposed in [53], however, works only for multiple TCP flows. Approaches like [152] assume that the inter-arrival time of the packets is exponentially distributed, and propose a queueing model for estimating end-to-end delay. However, this method focuses on modeling network nodes for computing the required parameters for network design and planning, and does not propose a method for real-time delay monitoring.

In this chapter, we propose a passive method to measure link delay in SDN networks, without making any assumptions about the traffic distribution. The proposed method solves the issues stated earlier. It depends entirely on the queue statistics messages from the switches and, as a result, has zero data-plane footprint. Monitoring overhead is re-

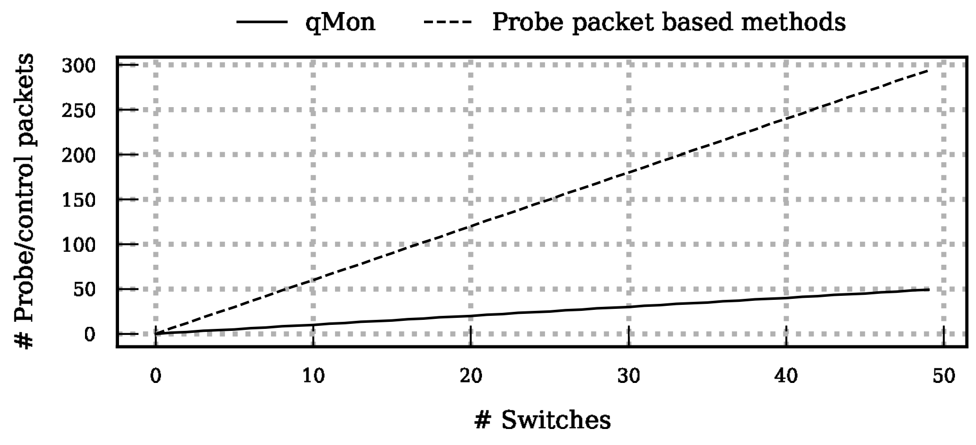


Figure 4.2: Number of switches vs Number of probe/control packet

duced considerably as estimation of queueing delay only requires an application of the Little's formula over the queue statistics received. Delay calculation can, therefore, be implemented inline with other QoS related modules, and does not require a dedicated controller module. The controller is now exempt from creating probe packets and receiving them, thus considerably reducing the processing overhead. Queue statistics message received from a switch includes the queueing information for all the queues at each of the ports. Therefore, queueing delay for each of the queues can be computed with the information from a single queue statistics message. Thus, the proposed method is scalable, as it requires sending of a single queue-statistics request message to each of the switches to monitor queueing delay at all the links in the network. This is in contrast to the existing active probe packet based methods which require sending a probe packet to each of the output port queues in all the switches to measure the queueing delay. Figure 4.2 shows the number of probe packets injected by active probe based methods versus the number of control packets required in qMon to measure the queueing delay at all the output ports for a given number of switches in the network.

4.3 System Model

We apply queueing theory to model the flow of packets through Open vSwitch [54] on a linux datapath. Open vSwitch supports different datapaths on different platforms: Linux upstream, Linux OVS tree, Userspace, Hyper-V (Windows datapath) [153]. However, only Linux upstream and Linux OVS tree support QoS-shaping. For evaluating the current work, we worked on Linux OVS tree datapath, provided by Open vSwitch v2.11.1.

4.3.1 Delays

We define link delay as the time required to transmit a packet in its egress path, which is the sum of processing, queueing, transmission and propagation delay. Throughout this chapter, we assume, that processing, transmission, and propagation delays remain constant. Most of the modern routers are capable of processing the packets at almost the line rate. In SDN networks, a packet may not match any flow entry and is consequently sent to the controller (slow path). Although such packets experience larger processing delays than usual, they are usually the first packet of a flow and help the controller in establishing flow entries on the switches in the flow path. Therefore delay experienced by such packets is not representative of the delays experienced by majority of the packets that match the flow entries and are forwarded through the fast path. It is therefore assumed that each packet takes almost the same amount of time to get processed. Transmission delay is the time required to inject all the bits of a packet so that it can be transmitted over the physical medium, and is a function of the packet length. Assuming that the packet length remains constant on an average, transmission delay can also be assumed to be constant. Propagation delay is a function of the path length, which remains constant for a given system configuration.

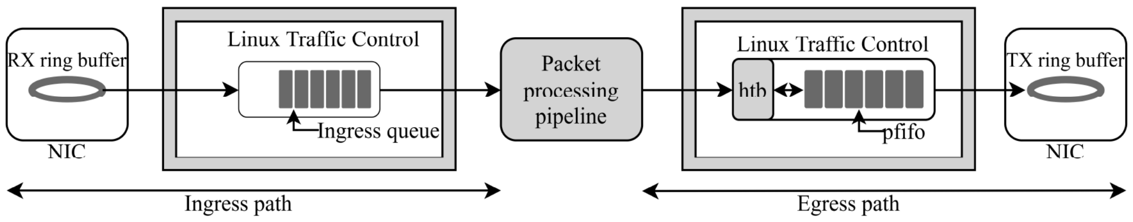


Figure 4.3: Queues in the ingress and egress path of the packet.

Queueing delay is the time for which a packet is enqueued in a queue before it is transmitted. Figure 4.3 illustrates the ingress and egress path of a packet through Open vSwitch on a linux datapath: packets are queued at the Network Interface Card (NIC) ring buffer on receive (RX) path, ingress queue of the RX interface, egress queue of the transmit (TX) interface and the NIC ring buffer on the TX path. The NIC ring buffer is also known as the driver queue or the direct memory access (DMA) ring buffer and is a part of the NIC. The ingress and egress queues are a part of the Traffic Control (TC) subsystem of the linux kernel [154, 155]. TC allows traffic shaping at the egress queues, while only traffic policing is possible at the ingress queues. It is assumed that for most of the time, the CPU is capable of processing the packets at the input port so that there is no buffering at the ingress queue. It is also assumed that the TC egress queues are configured so that the total bandwidth allocated to them does not exceed the line-rate of the underlying physical layer. Under this assumption, packets are buffered at the TC queues and no buffering takes place at the egress driver queue. Link latency due to buffering at the TC queues is a function of the traffic distribution arriving at the queues. The next section summarises the queueing of packets in linux traffic control and the linux-htb queueing discipline to illustrate the same.

4.3.2 Queueing in Traffic Control and linux-htb

The linux TC consists of various queueing disciplines (qdiscs) which fall under two categories: classless and classful. The default queueing discipline in the traffic control is pfifo_fast [156], which can be changed through userspace TC utility. Classful qdiscs are flexible, that is, classful or classless child qdiscs can be attached to them, and can share bandwidth with other classful qdiscs, when possible. Leaf classes have a classless qdisc attached to them. The queues managed by the classless qdiscs (attached to classful qdiscs) is where the packets finally get enqueued or dequeued, by the algorithm corresponding to that class. Examples of classful qdiscs are HTB (Hierarchical Token Bucket) and CBQ (Class Based Queueing) [157]. Classless qdiscs are elementary qdiscs and are rigid in the sense that they cannot have children, nor can they share bandwidth with other classes. They maintain a queue, from which the packets get enqueued or dequeued by the algorithm corresponding to the qdisc. Examples of classless queueing disciplines are: pfifo, bfifo, Token Bucket Filter (TBF), Stochastic Fairness Queueing (SFQ) , pfifo_fast (default used by linux TC) [157].

Linux-htb (from now on referred to as HTB) [158] is a classful queueing discipline, which means child qdiscs can be attached to it, which forms a tree like structure. Each HTB class has a token bucket associated with it, which are filled with tokens at a rate which is determined by the rate assigned to the class. To dequeue a packet, HTB charges certain amount of tokens proportional to the size of the packet from the bucket associated with the class. If there are not enough tokens, then the class tries to borrow tokens from the sibling classes. If enough tokens are not available, the packet has to wait until the bucket has enough tokens. Thus, bandwidth is limited by throttling the packets, which is also known as shaping.

When a packet originating from a child class is dequeued from its parent class, a number of tokens depending on the size of the packet is charged from the bucket of the parent class. These tokens then will not be available for packets originating from other child classes. This ensures that child classes cannot have a rate more than their parent class. While enqueueing and dequeueing, the kernel interacts only with the root qdisc. To enqueue a packet, kernel calls `htb→enqueue()` located in `/net/sched/sch_htb.c`, which further calls `htb→htb_classify()` to classify the packet into one of the child HTB classes. It walks the tree and enqueues to a classless qdisc attached to one of the leaf nodes. To dequeue a packet, the kernel calls `dequeue()` on the root qdisc, which calls dequeue function of the child classes, and so on, until a packet is dequeued from the leaf class, and is passed on to the parent classes, charging tokens from the buckets in the process, until it is dequeued from the root qdisc. The dequeued packet is then placed onto the driver queue for transmission. Open vSwitch does not support multiple levels of hierarchy in classful queueing disciplines. It is therefore only possible to configure multiple HTB queues at the first level. Borrowing of tokens can still take place among the queues at the first level. By default, the leaf nodes have a pfifo qdisc attached to them, with a queue length of `txqeueuelen`, which is a parameter associated with each interface. It has a default value of 1000 packets, which can be changed with the `iproute2` userspace utility. The queue length plays a significant role during burst traffic, i.e., when there is a large amount of traffic in a short period of time, this can lead to queues getting filled up in a very short span of time. If a queue gets filled up beyond its maximum size then the incoming packets are dropped. Throughout the chapter, we have assumed a single queue at each of the output port.

4.3.3 Queueing Model

Queueing delay is a function of the packet arrival and service rate distributions. Due to the nature of the packet scheduler (HTB), the service rate distribution is a function of the packet size. Internet traffic, however, is dynamic in nature which makes it tough to categorize the traffic into a known distribution [159]. Therefore, our objective is to model the TC queues as a single queue and a single server queueing problem, and solve it for a general packet arrival and service rate distribution using Little's Law,

$$L = \lambda W \tag{4.1}$$

where L is the average number of packets in the queue, λ is the average arrival rate, and W is the average waiting time.

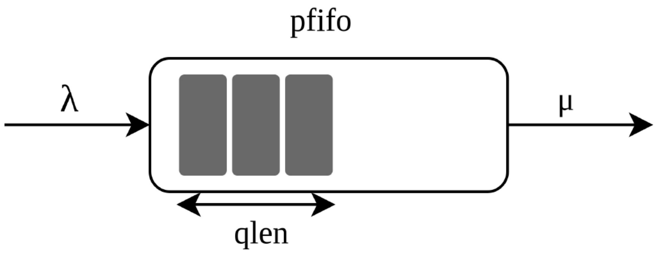


Figure 4.4: A TC queue

Figure 4.4 shows a TC queue modelled as a single queue, single server queueing system. To meet end-to-end QoS requirements of latency-sensitive applications, it becomes necessary to estimate delay at short intervals of time. So that the routing decisions can be taken to adapt the current load conditions and QoS requirements. Let the polling interval for the queue-statistics be t. Let t_1 and t_2 be two consecutive polling epochs. Our objective is to obtain the average waiting time of packets in the finite time interval [t_1,t_2], where t_2 - t_1 = t.

Given λ_{av} as the average rate of arrival of packets in the queueing system and average queue length ($qlen_{av}$) as the average number of packets in the system over infinitely large time, the average waiting time for packets in the queue can be obtained by using Little's law as $\overline{W}$ = qlen$_{av}/\lambda_{av}$. The equality however does not necessarily hold true over finite time intervals due to interval edge effects [160]. Let $\overline{qlen}(t_1, t_2)$ be the mean queue length, $\overline{\lambda}(t_1, t_2)$ be the mean arrival rate of the packets, and $\overline{W}_{\overline{qlen},\overline{\lambda}}(t_1, t_2)$ be the mean waiting time of a packet in the queue in the interval $[t_1, t_2]$. Authors in [160] regard $\overline{qlen}(t_1, t_2)/\overline{\lambda}(t_1, t_2) = \overline{W}_{\overline{qlen},\overline{\lambda}}(t_1, t_2)$ as an estimator of the underlying true average. They also suggest taking a statistical approach to estimate the waiting times, and using the method of batch means to apply the estimator $\overline{W}_{\overline{qlen},\overline{\lambda}}(t_1, t_2)$ to estimate a confidence interval for $\overline{W}(t_1, t_2)$. Section 4.3.5 discusses how to apply batch means method.

4.3.4 Estimator for $\overline{W}(t_1, t_2)$

Our estimation of waiting time relies on queue statistics obtained by sending an OpenFlow queue-stats request message from controller to the switches. OpenFlow v1.3 queue-stats reply message has a counter of the number of transmitted packets (*tx_packets*) through the queue. Let t_1 and t_2 be two consecutive times at which the controller receives the queue statistics. For a stable system, $\overline{\lambda}(t_1, t_2)$ is equal to the mean throughput over the interval $[t_1, t_2]$, and can be estimated using equation 4.2.

$$\overline{\lambda}(t_1, t_2) = (tx\_packets_{t_2} - tx\_packets_{t_1})/(t_2 - t_1) \tag{4.2}$$

Queue statistics reply message however does not contain any field with the queue length information. We observed that the current length of each queue is maintained in the TC subsystem of the kernel. By modifying `netdev-linux.c` in Open vSwitch 2.11, queue statistics reply message can be modified to include queue length. As such, the queue statistics message reply after modification contains the following fields: $\langle queue\_id, tx\_bytes, tx\_packets, qlen, errors, time\rangle$. $\overline{qlen}(t_1, t_2)$ can then be obtained in the following way:

$$\overline{qlen}(t_1, t_2) = (qlen_{t_1} + qlen_{t_2})/2 \tag{4.3}$$

Where, $qlen_{t_1}$ and $qlen_{t_2}$ are the queue lengths at time t_1 and t_2 respectively.

Estimator for $\overline{W}$, in the interval $[t_1, t_2]$ can then be obtained by Little's law as:

$$\overline{W}_{\overline{qlen},\overline{\lambda}}(t_1, t_2) = \overline{qlen}(t_1, t_2)/\overline{\lambda}(t_1, t_2) \tag{4.4}$$

4.3.5 Confidence Intervals for $\overline{W}(t_1, t_2)$

Consider m consecutive estimation intervals $[t_0, t_1], [t_1, t_2], ..., [t_{m-1}, t_m]$. Let the estimated waiting times in the intervals be $\overline{W}_{\overline{qlen},\overline{\lambda}}(t_0, t_1)$, $\overline{W}_{\overline{qlen},\overline{\lambda}}(t_1, t_2)$, ..., $\overline{W}_{\overline{qlen},\overline{\lambda}}(t_{m-1}, t_m)$. Confidence intervals for the mean waiting time over the union of the intervals $[t_0, t_1] \cup [t_1, t_2] \cup ... \cup [t_{m-1}, t_m] = [t_0, t_m]$, $\overline{W}(t_0, t_m)$ can be calculated by applying the formulas given by [160]. We consider batches of size m = 5, by taking m consecutive intervals. The 90% confidence interval can then be calculated by formulas (24) in [160].

$$\begin{aligned}
&\overline{W}^{(m)}_{\overline{qlen},\overline{\lambda}}(t_0, t_m) = \frac{1}{m}\sum_{k=1}^{m} \overline{W}_{\overline{qlen},\overline{\lambda}}(t_{k-1}, t_k),\\
&\overline{S}_{(m)}(t_0, t_m) = \frac{1}{m-1}\sum_{k=1}^{m}(\overline{W}_{\overline{qlen},\overline{\lambda}}(t_{k-1}, t_k) - \overline{W}^{(m)}_{\overline{qlen},\overline{\lambda}}(t_0, t_m))^2,\\
&\left[\overline{W}^{(m)}_{\overline{qlen},\overline{\lambda}}(t_0, t_m) - \frac{t_{0.025,m-1} S_m(t_0, t_m)}{\sqrt{m}}, \overline{W}^{(m)}_{\overline{qlen},\overline{\lambda}}(t_0, t_m) + \frac{t_{0.025,m-1} S_m(t_0, t_m)}{\sqrt{m}}\right]
\end{aligned} \tag{4.5}$$

The corresponding formulas for the current use case is given by Equation (4.5) above. It is based on Student's t-distribution. For $t_{0.025,m-1}$ = $t_{0.025,4}$ = 2.132.

4.3.6 Calculation of Calibration Constant and Delay

Assuming that packet's length and the system configuration remain constant; processing delay, transmission delay and propagation delay remain constant and can be calculated as constants by sending probe packets through the link. Let $\overline{D}_{\overline{qlen},\overline{\lambda}}(t_1, t_2)$ be the estimator for the mean link delay $\overline{D}(t_1, t_2)$ in the interval $[t_1, t_2]$. $\overline{D}_{\overline{qlen},\overline{\lambda}}(t_1, t_2)$ is given by Equation (4.6).

$$\overline{D}_{\overline{qlen},\overline{\lambda}}(t_1, t_2) = \overline{W}_{\overline{qlen},\overline{\lambda}}(t_1, t_2) + C \tag{4.6}$$

where C is the calibration constant. The confidence interval for $\overline{W}(t_1, t_2)$ and hence, $\overline{D}_{(t_1,t_2)}$ can be found out by batch interval method as discussed.

For demonstration purpose, consider Figure 4.1. To measure calibration constant, C, for the link $S_1 - S_2$ a probe packet is sent from controller (C_0) to switch S_1. Flow entry at switch S_1 sends the probe packet on the link $S_1 - S_2$. From switch S_2 the packet is sent back to the controller. The total time taken by the probe packet to traverse C_0 −

$S_1 - S_2 - C_0$ is calculated by subtracting the arrival and the departure time of the packet at the controller. To obtain the calibration constant for the link $S_1 - S_2$ the control link delays, $C_0 - S_1$ and $C_0 - S_2$, are subtracted from the total transit time of the probe packet. Control link delay between the controller and a switch is calculated by sending a statistics request message from the controller to the switch. The time elapsed between sending of the stat request message and receiving the corresponding reply at the controller is halved (assuming symmetric delays) to get the one way delay on control link. The control link delays, $C_0 - S_1$ and $S_2 - C_0$, are subtracted from the total time to traverse $C_0 - S_1 - S_2 - C_0$ to get the delay on link $S_1 - S_2$. The calibration constant is obtained by averaging over the delays obtained from several iterations of the method discussed above. Similar procedure is used to obtain the calibration constant for the experimental topology.

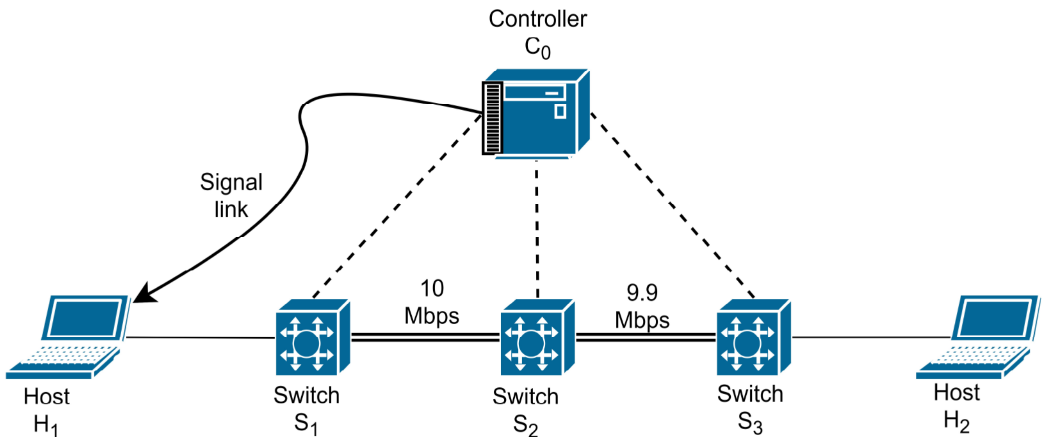


Figure 4.5: A topology with 3 switch, 2 hosts and controller connected in a linear fashion for evaluating the proposed method.

4.4 Experimental Setup and Evaluation Criteria

We evaluate the performance of qMon on a testbed consisting of three systems installed with the patched version of Open vSwitch v2.11.1 connected in a linear topology. A host is connected to each edge switch, and all the switches are directly connected to the controller. The topology for the testbed is shown in Figure 4.5. The link bandwidth is set to 10 Mbps on link S_1-S_2 and 9.9 Mbps on link S_2-S_3. Bandwidth for link S_2-S_3 is less than link S_1-S_2 to ensure buffering of packets on link S_2-S_3. Each of the systems has the following capabilities - Processor: Intel(R) Core(TM) i5-4590 CPU @3.30GHz, RAM: 32

GiB (8 GiBx4), OS: Ubuntu 18.04 LTS (Linux 4.15.0-91-generic), NIC: 1 Gbits/s between two switches or hosts, 100 Mbits/sec between controller and switches.

We use D-ITG [138], to generate traffic from host h1 to host h2. For each experiment, the traffic rate is set at a value to enable moderate buffering at TC queues. The evaluation is performed over two kinds of traffic distributions: bursty with exponential ON/OFF periods and Poisson with exponentially distributed packet length. For bursty traffic, packet length is set to 512 bytes with varying ON/OFF ratio in different experiments. The experiments are named as $B - X - Y$, where B means bursty traffic, X is ON time, and Y is OFF time in ms. For Poisson traffic, mean packet length is set to 930 bytes and 512 bytes for experiments `pois-1` and `pois-2` respectively. However, since RTT measurements are running parallelly, the observed mean packet size and arrival rates are 936 and 1362, 525 and 2362 for `pois-1`, and `pois-2` experiments respectively.

The delay per link is estimated using qMon. The round trip time is estimated by adding the estimated link latencies at the links $S_1 - S_2, S_2 - S_3, S_3 - S_2, S_2 - S_1$. The delay trends obtained from the qMon RTT are compared with ping RTT measurements. Ping has been used as a standard tool for RTT measurements in IP networks. It is therefore a reliable measure of RTT. Although ping measurements are active RTT measurements, it will not change the traffic behavior in the current setup, since a traffic generator is being used to generate traffic of a fixed distribution. It is complicated to compare two-time series data due to the high frequency of variations in measurements. It is, therefore, desirable to visualize and compare the low-frequency component of the delay variations. We consider the delay measurements as discrete signals and apply a low pass filter to the delay signals. This will help us to compare the low-frequency components of qMon and ping RTT signals.

We use `windowed-sinc` as our low pass filter [161]. For constructing the filter, two parameters are required: cut-off frequency(f_c) as a fraction of the sampling rate and transition band (b) as a fraction of the sampling rate. The length of the filter N can be determined by the formula (Equation 4.7).

$$N = 4/b \tag{4.7}$$

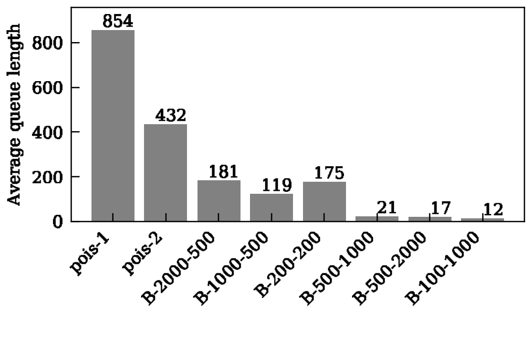
The low frequency signals are obtained by convulsing the sinc function(c_0(n)) with the delay signal. The sinc function is constructed as shown in Equations 4.8, 4.9, and 4.10.

$$c_1(n) = sinc(2 * f_c * (n - (N-1)/2)) \tag{4.8}$$

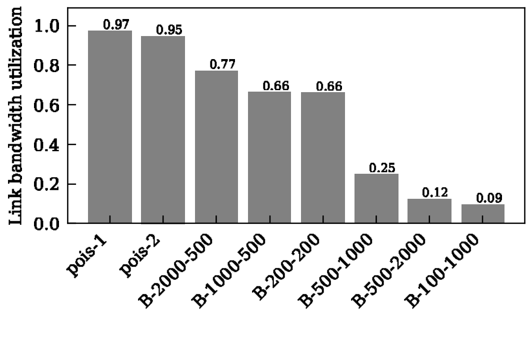
$$\begin{aligned} window(n) = 0.42 - 0.5 * cos(2 * pi * n/(N-1)) \\ +0.08 * cos(4 * pi * n/(N-1)) \end{aligned} \tag{4.9}$$

$$c_0(n) = c_1(n) * window(n) \tag{4.10}$$

We set the cut-off frequency (f_c) at 1% of the sampling rate and the transition bandwidth (b) at 8% of the sampling rate. Quantitative evaluation of the trend match between qMon and ping RTTs has been done by comparing Pearson's correlation coefficient between the low-pass delay signals. Pearson's correlation coefficient measures the linear synchrony between two signals. Values between -1 and 0 indicate a negative correlation, 0 and +1 indicate a positive correlation. In the current use case, it is already known that there is no implicit co-relation between qMon and ping RTT signals. However, the correlation values are an indication of the similarity between the signals. Figures 4.6 (a), and (b) compare the average queue length and average link bandwidth utilization for the experiments conducted.



(a)



(b)

Figure 4.6: (a) Average queue length (b) Average link bandwidth utilization (ρ) for the traffic scenarios considered.

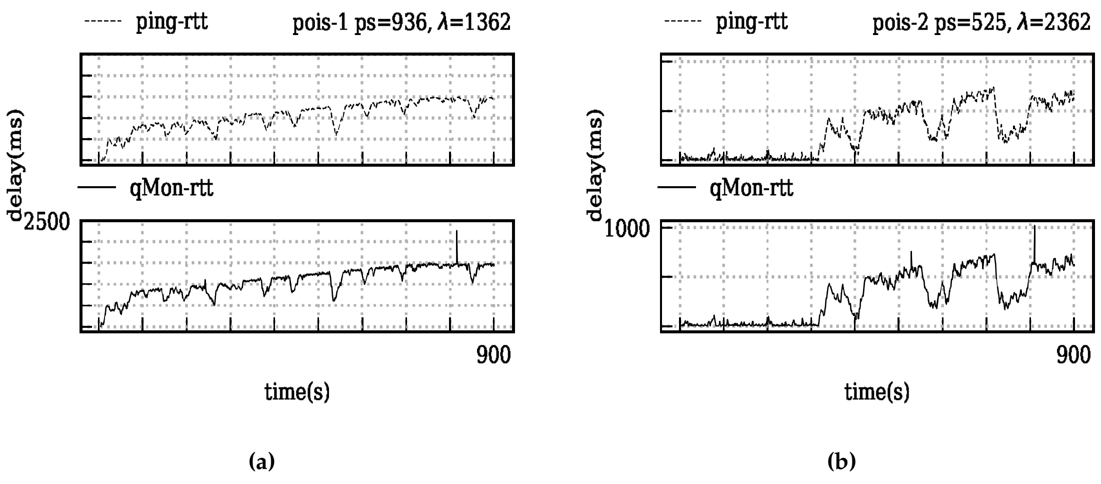


Figure 4.7: qMon versus ping round trip times (RTT) for poisson traffic, ps = packet size, λ = mean packet arrival rate

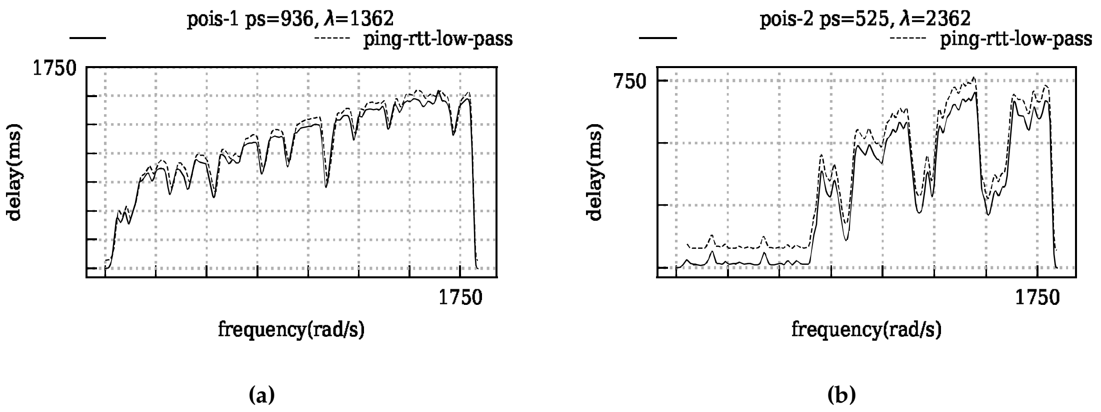


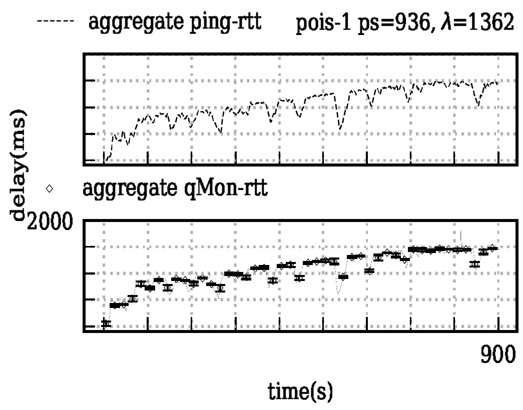
Figure 4.8: Low pass delay signals for qMon versus ping RTT for poisson traffic, ps = packet size, λ = mean packet arrival rate. Ping RTT signal is offset by +70ms for better visibility.

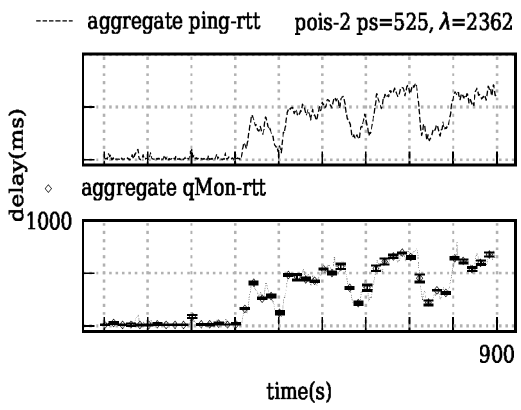
4.5 Results

Results have been organised into two sections. Section 4.5.1 discusses the delay trends between `qMon` and `ping` RTT for poisson traffic. Section 4.5.2 discusses the same for bursty traffic.

4.5.1 Poisson traffic

Figure 4.7 (a) and (b) show the graph of estimated RTT value with respect to the time, for two different types of Poisson traffics (`pois-1`, `pois-2`), using, qMon and ping-rtt methods. It can be observed from Figure 4.8 (a) and (b) that low frequency qMon and ping RTT signals in traffics `pois-1` and `pois-2` closely coincide. The ping RTT signals





(a) (b)

Figure 4.9: 90% confidence interval for qMon RTT for poisson traffic, ps = packet size, λ = mean packet arrival rate.

in Figure 4.8 have been offset by +70ms for better visibility of qMon and ping RTT signals.

Table 4.1 shows that the correlation values for `pois-1` and `pois-2` traffics are quite high ($\geq$ .99). The trend match can be attributed to the fact that queueing system is in steady state. Since the input traffic rate is Poisson distributed and packet sizes are exponentially distributed, the queueing system can be approximated as M/M/1/K with K (buffer capacity) = 1000 packets. The queueing system therefore reaches a steady state with Poisson distributed traffic. Smaller value of 90% confidence intervals in Figure 4.9 (a), and (b) also indicate that average waiting time estimations are fairly accurate in case of Poisson traffic.

Table 4.1: Pearson co-efficient for all the experiments

Experiment Name	**Correlation coefficient**
pois-1	.99
pois-2	.99
B-2000-500	.97
B-1000-500	.87
B-200-200	.84
B-500-1000	.60
B-500-2000	.74
B-100-1000	.32

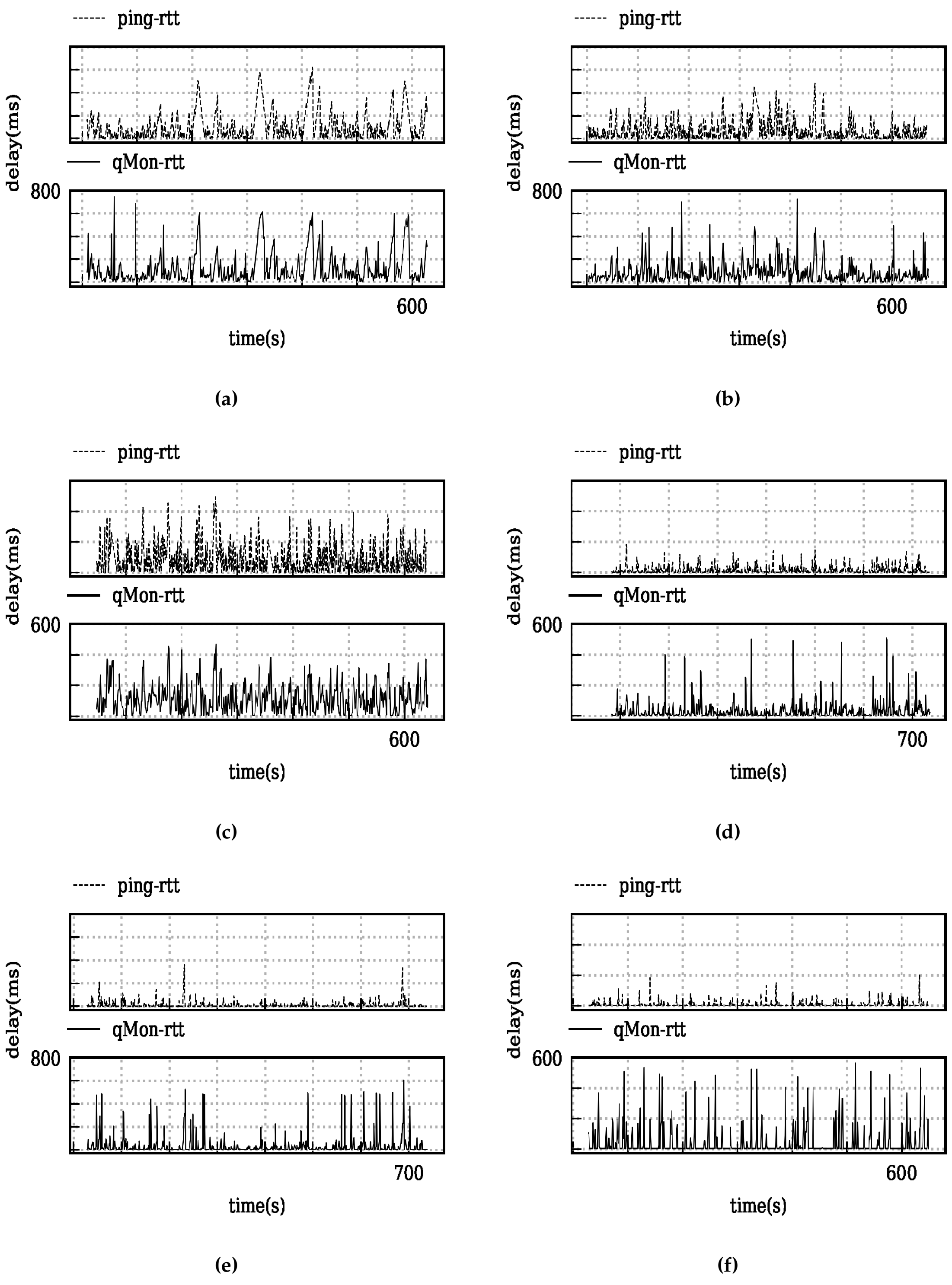
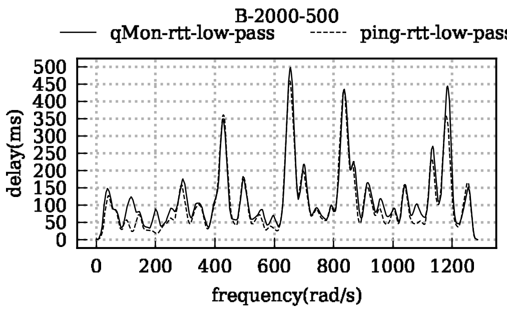


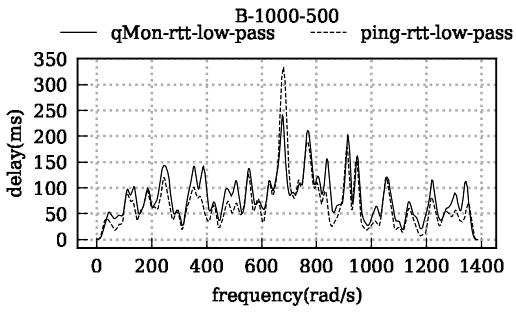
Figure 4.10: qMon versus ping round trip times (RTT) for bursty traffic

4.5.2 Bursty traffic

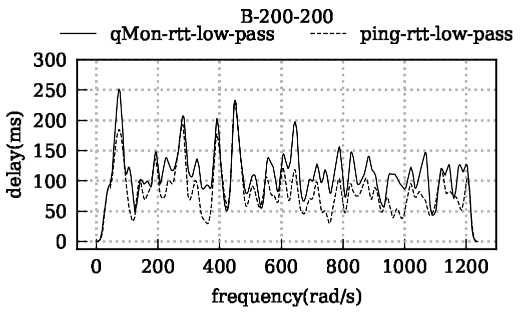
Figure 4.10 (a) - (f) illustrate the delay vs time plot for the RTT obtained by qMon and ping rtt methods for bursty traffic. It can be observed from Figure 4.11 (a) and (b) that



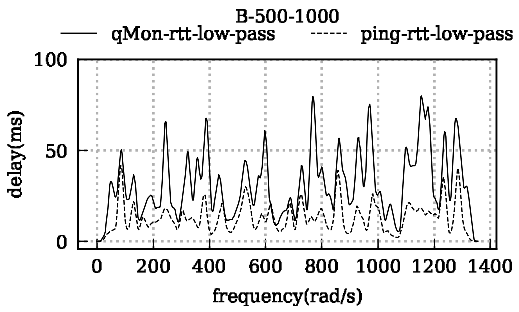
(a)



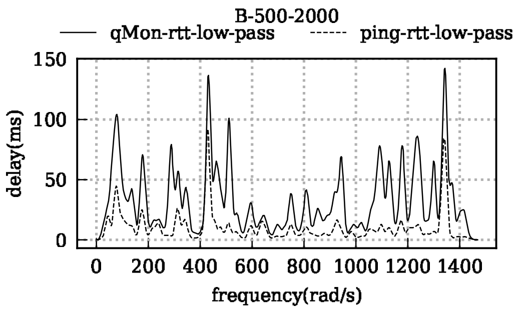
(b)



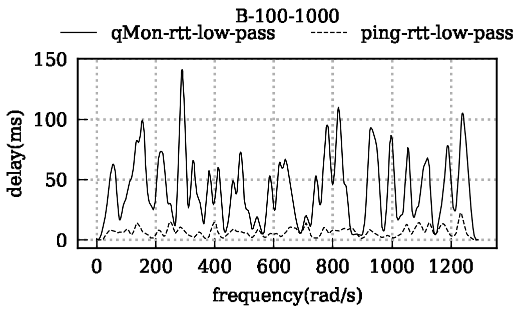
(c)



(d)



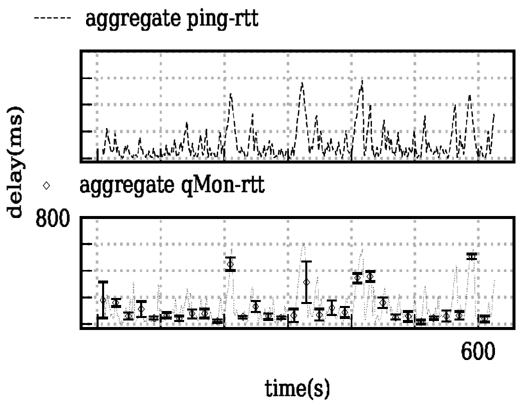
(e)



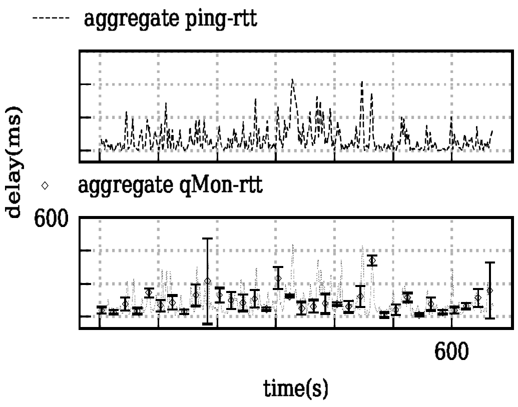
(f)

Figure 4.11: Low pass delay signals for qMon versus ping RTT for bursty traffic.

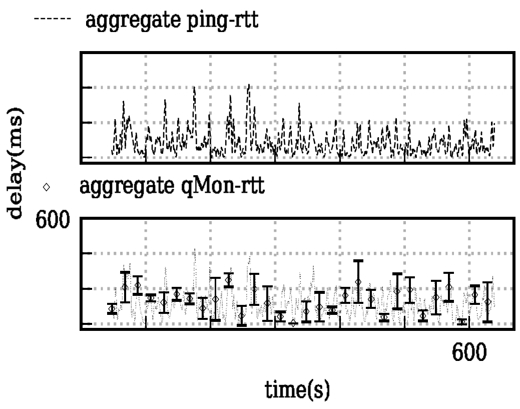
low frequency qMon and ping RTT signals in `B-2000-500` and `B-1000-500` closely coincide. Table 4.1 shows that the correlation values for `B-2000-500` and `B-1000-500` are quite high. For `B-200-200`, `B-500-1000`, `B-500-2000` and `B-100-1000`, the correlation values decrease with a decrease in the ratio of ON/OFF periods, with the exception of `B-500-2000`. High correlation values for `B-2000-500` and `B-1000-500` can be attributed to the steady state of the queueing system due to generation of traffic at a constant rate during ON periods. Since the ON periods are of larger duration than OFF periods for `B-2000-500` and `B-1000-500`, the queue is in a steady state for a larger duration leading to accurate waiting time estimations. As the ratio of ON/OFF period decreases, the



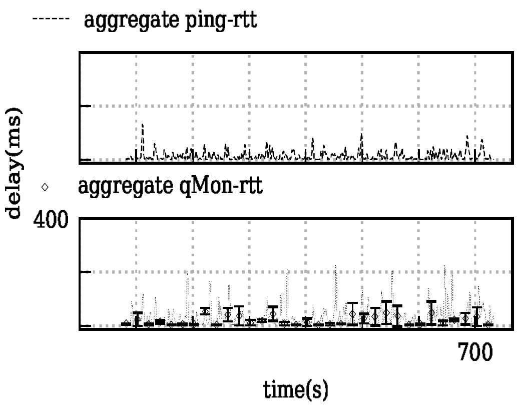
(a)



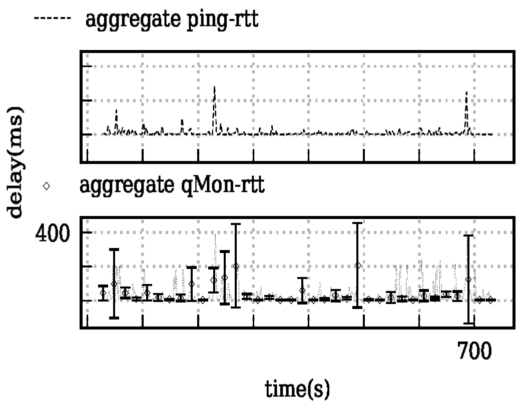
(b)



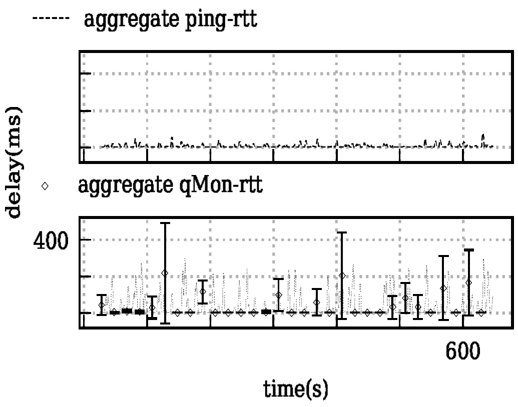
(c)



(d)



(e)



(f)

Figure 4.12: 90% confidence interval for qMon RTT for bursty traffic.

duration for which the queue is in a steady state decreases (Table 4.1), with the exception of `B-500-2000`. Higher correlation value of `B-500-2000` than `B-500-1000` may be a result of (1) same length of ON periods and measurement interval (500ms), which might lead to synchronization of OFF periods and the measurement epochs. Probability of this happening is less because ON/OFF periods are exponentially distributed, and (2) Longer mean OFF period (2000ms) in `B-500-2000` than in `B-500-1000` (1000ms). Longer OFF period lead to steadiness in the system, leading to higher correlation between estimated and measured RTT. Low correlation value in `B-100-1000` is expected as the ON period (100ms) is very less as compared to the measurement interval (500ms), because (1) Even though the ON/OFF periods are exponentially distributed, because of very low mean ON period (100ms) there is a high probability of measurement epochs synchronizing with the OFF periods. (2) Even when the measurement intervals are synchronized with the ON period, the burst of traffic happens only for 100ms, but the average of all burst packets is taken over a time interval of 500ms, leading to a inaccurate and numerically low estimated TX_rate. The numerically low estimated TX_rate leads to very high estimated values of waiting time by Little's Law. This is apparent on observing Figure 4.11(f). Figure 4.12(a)-(f) show the 90% confidence interval plots for experiments with bursty input. Confidence interval plots indicate a similar trend. `B-2000-500` and `B-1000-500` have narrow confidence intervals on an average as compared to `B-200-200`, `B-500-1000`, `B-500-2000` and `B-100-1000`.

4.6 Implementation Details

The proposed queueing delay monitoring approach, qMon, requires a slight modification of Open vSwitch v2.11.1 and Ryu running on Ubuntu 18.04 LTS as the underlying operating system. Open vSwitch leverages `RTNETLINK` to communicate with the Traffic Control (TC) subsystem of the Linux kernel to add/remove or get a queueing discipline [162]. To get queue-statistics, ovs switch makes a `RTNETLINK` request message (`RTM_GETQDISC`). The corresponding `RTNETLINK` reply received from the kernel is parsed into `struct gnet_stats_basic` and `struct gnet_stats_queue` defined in `linux/gen_stats.h`. The sturucture is given below for reference,

```
(linux/gen_stats.h)

/**
 * struct gnet_stats_basic - byte/packet throughput statistics
 * @bytes: number of seen bytes
 * @packets: number of seen packets
 */
struct gnet_stats_basic {
__u64 bytes;
__u32 packets;
};

/**
 * struct gnet_stats_queue - queuing statistics
 * @qlen: queue length
 * @backlog: backlog size of queue
 * @drops: number of dropped packets
 * @requeues: number of requeues
 * @overlimits: number of enqueues over the limit
 */
struct gnet_stats_queue {
__u32 qlen;
__u32 backlog;
__u32 drops;
__u32 requeues;
__u32 overlimits;
};
```

```
(ovs/include/openflow/openflow-1.3.h)

/* Body of reply to OFPMP13_QUEUE request */
struct ofp13_queue_stats {
    struct ofp11_queue_stats qs;
    ovs_be32 duration_sec;     /* Time queue has been alive in seconds. */
    ovs_be32 duration_nsec;    /* Time queue has been alive in nanoseconds
                                  beyond duration_sec. */
};
```

The struct netdev_queue_stats defined in `ovs/lib/netdev.h` represents queue-stats reply. OpenFlow 1.3 queue-stats reply does not have a field for queue length. Open vSwitch v2.11.1 therefore does not include a field for queue length in struct netdev_queue_stats.

We observed that for experimental purposes, we can add a queue length field (‘`uint64_t qlen`’) to `struct netdev_queue_stats`. The field `qlen` is populated with queue length received in the `struct gnet_stats_queue` and is propagated up to the controller as the queue-statistics reply. Structs that have changed are given below,

```
---------------
Before: (ovs/include/openvswitch/ofp-queue.h)
------------------------------
struct ofputil_queue_stats {
    ofp_port_t port_no;
    uint32_t queue_id;

    /* Values of unsupported statistics are set to all-1-bits (UINT64_MAX).
        */
    uint64_t tx_bytes;
    uint64_t tx_packets;
    uint64_t tx_errors;

    /* UINT32_MAX if unknown. */
    uint32_t duration_sec;
    uint32_t duration_nsec;
};
---------------
After: (ovs/include/openvswitch/ofp-queue.h)
------------------------------
struct ofputil_queue_stats {
    ofp_port_t port_no;
    uint32_t queue_id;

    /* Values of unsupported statistics are set to all-1-bits (UINT64_MAX).
        */
    uint64_t tx_bytes;
    uint64_t tx_packets;
    uint64_t tx_errors;
    /* qlen parameter */
    uint64_t qlen;

    /* UINT32_MAX if unknown. */
    uint32_t duration_sec;
    uint32_t duration_nsec;
};
--------------------------------------------------------------------



---------------
Before: (ovs/lib/netdev.h)
------------------------------
struct netdev_queue_stats {
    /* Values of unsupported statistics are set to all-1-bits (UINT64_MAX).
        */
    uint64_t tx_bytes;
    uint64_t tx_packets;
    uint64_t tx_errors;

    /* Time at which the queue was created, in msecs, LLONG_MIN if unknown.
        */
    long long int created;
};
---------------
After: (ovs/lib/netdev.h)
------------------------------
struct netdev_queue_stats {
    /* Values of unsupported statistics are set to all-1-bits (UINT64_MAX).
        */
    uint64_t tx_bytes;
    uint64_t tx_packets;
    uint64_t tx_errors;
    /* qlen parameter */
    uint64_t qlen;

    /* Time at which the queue was created, in msecs, LLONG_MIN if unknown.
        */
    long long int created;
};

--------------------------------------------------------------------
```

```
---------------
Before: (ovs/include/openflow/openflow-1.1.h)
------------------------------
struct ofp11_queue_stats {
    ovs_be32 port_no;
    ovs_be32 queue_id;         /* Queue id. */
    ovs_be64 tx_bytes;         /* Number of transmitted bytes. */
    ovs_be64 tx_packets;       /* Number of transmitted packets. */
    ovs_be64 tx_errors;        /* # of packets dropped due to overrun. */
};
---------------
After: (ovs/include/openflow/openflow-1.1.h)
------------------------------
struct ofp11_queue_stats {
    ovs_be32 port_no;
    ovs_be32 queue_id;         /* Queue id. */
    ovs_be64 tx_bytes;         /* Number of transmitted bytes. */
    ovs_be64 tx_packets;       /* Number of transmitted packets. */
    ovs_be64 tx_errors;        /* # of packets dropped due to overrun. */
    ovs_be64 qlen;             /* qlen parameter */
};
```

Since the queue-stats reply is modified to include queue length, minor modifications were made to Ryu controller, so that it can recognize the new field as queue length. Accordingly, the following minor changes were made to Ryu,

```
In file 'ryu/ofproto_v1_3.py':
The struct pack string and the size of the struct were changed:
---------------
Before:
------------------------------
OFP_QUEUE_STATS_PACK_STR = 'IIQQQII'
OFP_QUEUE_STATS_SIZE=40
---------------
After:
------------------------------
```

```
OFP_QUEUE_STATS_PACK_STR = 'IIQQQQII'
OFP_QUEUE_STATS_SIZE=48


In file 'ryu/ofproto_v1_3_parser.py':
added 'qlen' to 'ofproto_parser.namedtuple()' for 'OFPQueueStats', so that
    the new field is recognized as 'qlen':
---------------
Before:
------------------------------
class OFPQueueStats(ofproto_parser.namedtuple('OFPQueueStats', (
        'port_no', 'queue_id', 'tx_bytes', 'tx_packets', 'tx_errors',
        'duration_sec', 'duration_nsec'))):
---------------
After:
------------------------------
class OFPQueueStats(ofproto_parser.namedtuple('OFPQueueStats', (
        'port_no', 'queue_id', 'tx_bytes', 'tx_packets', 'tx_errors', 'qlen',
        'duration_sec', 'duration_nsec'))):
```

The delay monitoring module in the controller makes queue-statistics request messages to all the switches in the network at fixed intervals. We fixed the time interval to 500ms. For the time interval $[t_1, t_2]$, (where $t_2 - t_1$ = 500ms) the module uses the queue statistics received at time t_1 and t_2 to estimate the mean queueing delay for the interval $[t_1, t_2]$. The module can optionally store the estimated mean queueing delay for the last m intervals to calculate a 90% confidence interval for the union of these intervals. In [160] the authors suggest the value of m be equal to 5, which is the value considered in the evaluation section. The implementation of qMon does not require any modification in the packet processing pipeline of Open vSwitch. If the target switches are hardware switches, it only requires changes to the software/firmware of the switches to support qMon. Enterprise infrastructure administrators/engineers can request the switch vendors to customize the software/firmware of the switches to meet their needs for QoS optimizations.

4.7 Summary

In this chapter, we proposed an efficient passive method, qMon, to monitor the average queueing delay in OpenFlow networks. qMon has no data plane footprint as it solely depends on the queue statistics messages from the switches. The controller is now exempt from creating and processing probe packets, thus the monitoring overhead is considerably reduced. Queue statistics message received from a switch includes the queueing information for all the queues at each of the interfaces. Therefore, queueing delay for each of the queues can be computed with the information from a single queue statistics message. Thus, the proposed method is scalable with respect to number of switches, ports, and queues at ports. The experimental results show that RTT estimation made by qMon has a high similarity with ping-based estimation. The results show a high correlation between qMon and ping RTT values for Poisson traffic and bursty traffic with large ON intervals. However, the method does not work well for the bursty traffic with burst intervals smaller than the current polling interval. The proposed method can be used to improve QoS.

Using qMon the controller can measure queueing delay, but it has to poll queue statistics from underlying switches. Therefore, it is very important to determine an optimal polling rate which minimizes the monitoring overhead without compromising with the accuracy of collected statistics. In the next chapter, we address this issue and propose a method to determine optimal polling rate/frequency.