Enhancing Responsiveness and Scalability for OpenFlow Networks via Control-Message Quenching

Tie Luo*, Hwee-Pink Tan*, Philip C. Quan†, Yee Wei Law‡ and Jiong Jin‡
*Institute for Infocomm Research, A*STAR, Singapore
†Faculty of Mathematics, University of Cambridge, UK
‡Department of Electrical & Electronic Engineering, University of Melbourne, Australia
E-mail: {luot, hptan}@i2r.a-star.edu.sg, cq214@cam.ac.uk, {ywlaw,jjin}@unimelb.edu.au

***Abstract*—OpenFlow has been envisioned as a promising approach to next-generation programmable and easy-to-manage networks. However, the inherent heavy switch-controller communications in OpenFlow may throttle controller responsiveness and, ultimately, network scalability. In this paper, we identify that a key cause of this problem lies in flow setup, and propose a Control-Message Quenching (CMQ) scheme to address it. CMQ requires minimal changes to OpenFlow, imposes no overhead on the central controller which is often the performance bottleneck, is lightweight and simple to implement. We show, via worst-case analysis and numerical results, an upper bound of performance improvement that CMQ can achieve, and evaluate the average performance via experiments using a widely-adopted prototyping system. Our experimental results demonstrate considerable enhancement of controller responsiveness and network scalability by using CMQ, with reduced flow setup latency and elevated network throughput.**

I. Introduction

The advent of software-defined networking (SDN) [1] and OpenFlow [2] is ushering in a new era of networking with a clearly-decoupled network architecture: a programmable data plane and a centralized control plane. The data plane is tasked to flow-based packet forwarding and the control plane centralizes all intelligent control such as routing, traffic engineering and QoS control. The flow-based packet forwarding is realized by programmable (i.e., user-customizable) flow tables via an open interface called OpenFlow, which is the de facto communication standard protocol between the control and the data planes. The control plane is usually embodied by a central controller, providing a global view of the underlying network to upper applications. This architecture is depicted in Fig. 1.

The split architecture of SDN, gelled by OpenFlow, brings substantial benefits such as consistent global policy enforcement and much simplified network management. While the standardization process, steered by Open Networking Foundation consortium [3], is still ongoing, SDN and OpenFlow have won wide recognition and support from a large number of industry giants such as Microsoft, Google, Facebook, HP, Deutsche Telekom, Verizon, Cisco, IBM and Samsung. It has also attracted substantial attention from the academia where research has recently heated up on a variety of topics related to OpenFlow.

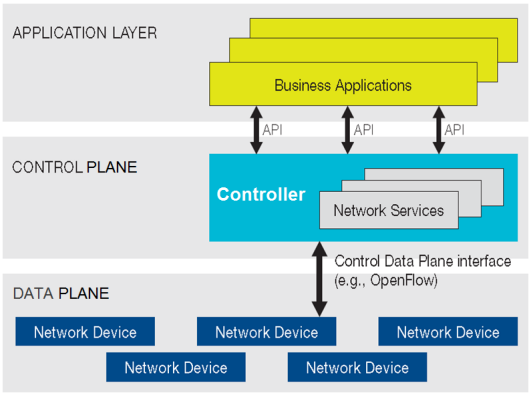


Fig. 1. The architecture of SDN [1].

In this paper, we address OpenFlow scalability by way of enhancing controller responsiveness. This was motivated by the imbalance between the flow setup rate offered by OpenFlow and the demand of real production networks. The flow setup rate, as measured by Curtis et al. [4] on a HP ProCurve 5406zl switch, is 275 flows per second. On the other hand, a data center with 1500 servers, as reported by Kandula et al. [5], has a median flow arrival rate of 100k flows per second, and an 100-switch network, according to Benson et al. [6], can have a spike of 10M flow arrivals per second in the worst case. This indicates a clear gap.

The inability of OpenFlow to meet the demanding requirements stems from its inherent design which lead to frequent *switch-controller communications*: to keep the data plane simple, OpenFlow delegates the task of routing (as required by flow setup) from switches to the central controller. As a result, switches have to consult the controller frequently for instructions on how to handle incoming packets, which taxes the controller's processing power and tends to congest switch-controller connections, thereby imposing a serious bottleneck to the scalability of OpenFlow.

Prior studies on the scalability issue focus on designing a distributed control plane [7], [8], devolving control

to switches [4], [9], [10] or end-hosts [11], or using multi-threaded controllers [12], [13]. These approaches either entail significant changes to OpenFlow (while a wide range of commercial OpenFlow products and prototypes have been off the shelf) or are complex to implement.

In this paper, we take a different and algorithmic approach, rather than architectural or implementation-based ones as used by above-mentioned work. We propose a Control-Message Quenching (CMQ) scheme to enhance controller responsiveness, in order to reduce *flow setup latency* and boost network throughput. By way of suppressing redundant control messages, it conserves more computational and bandwidth resources for the control plane and mitigates the controller bottleneck problem, thereby enhancing network scalability. CMQ is lightweight and easy to implement; it runs locally on switches without imposing any overhead on the controller. We show, via worst-case analysis and numerical results, an upper bound of control message reduction that CMQ can achieve, and evaluate its average performance via experiments using a widely-adopted prototyping system. The experimental results demonstrate the effectiveness of CMQ via considerably reduced flow setup latency and elevated network throughput.

The rest of the paper is organized as follows. Section II reviews the literature and Section III presents our proposed solution together with worst-case analysis. We evaluate the performance of CMQ in Section IV, and conclude the paper in Section V.

II. Related Work

Notwithstanding various advantages mentioned elsewhere, the architecture of a much-simplified data plane together with a centralized control planes of OpenFlow creates higher demand on switch-controller communications, thereby making the controller prone to bottleneck and throttling network scalability. To address this, Yu et al. proposed DIFANE [9] to push network intelligence down to the data plane. It partitions pre-installed flow-matching rules among multiple "authorized switches" and lets each authorized switch oversee one partition of the network; normal switches only communicate in the data plane with each other or authorized switches, while only authorized switches need to talk to the controller occasionally. However, the visibility of flow states and statistics is compromised as being largely hidden from the controller. It also requires significant changes to OpenFlow. Mahout [11] attempts to identify significant flows (called "elephant flows" therein) by end-hosts looking at the TCP buffer of outgoing flows, and does not invoke the controller for insignificant flows (called "mice" therein). The downside is that it requires modifying end-hosts, which is difficult to achieve because end-hosts are beyond the control of network operators. DevoFlow [4] (i) devolves control partly from the controller back to switches, an idea similar to [9] but incurs smaller changes to OpenFlow, using a technique called rule cloning, and (ii) only involves the controller for elephant flows, an idea similar to [11] but does not require modifying end-hosts. It also allows switches to make local routing decisions without consulting the controller, through multipath support and rapid rerouting. We take a different approach by improving the working procedures of the OpenFlow protocol and enhancing controller responsiveness; we only require minimal changes to switches and our proposed scheme is much simpler than DeveFlow.

Another line of thought to improve OpenFlow scalability focuses on redesigning the control plane. One such approach is to design a distributed control plane, as in Onix [7] and HyperFlow [8]. Another approach is to implement multi-threaded controllers, as done by Maestro [12] and NOX-MT [13]. Distributed control incurs additional overhead for synchronizing controllers' network-wide views, and increases complexity. Multi-threading is orthogonal to, yet compatible with, our approach, and hence can be applied to CMQ.

III. Control-Message Quenching

Our solution is based on the working procedures of the OpenFlow protocol, particularly the flow setup process. We identify that the major traffic occurred in switch-controller communications is comprised of two types of control messages: (1) `packet-in` and (2) `packet-out` or `flow-mod`. A `packet-in` message is sent by a switch to the controller in order to seek instructions on how to handle a packet upon *table-miss*, an event that an incoming packet fails to match any entry in the switch's flow table. The controller, upon receiving `packet-in`, will compute a route for the packet based on the packet header information encapsulated in the `packet-in`, and then respond to the requesting switch by sending a `packet-out` or `flow-mod` message: (i) if the requesting switch has encapsulated the entire original packet in `packet-in` (usually because of no queue), the controller will use `packet-out` which also encapsulates the entire packet; (ii) otherwise, if the `packet-in` only contains the header of the original packet, `flow-mod` will be used instead, which also only includes the packet header so that the switch can associate with the original packet (in its local queue). Either way, the switch can be instructed on how to handle the packet (e.g., which port to send to), and will record the instruction by installing a new entry in its flow table to avoid re-consulting the controller for subsequent packets that should be handled the same way.

This mechanism works fine when the switch-controller connection, referred to as *OpenFlow channel*

[14],[1] has ample bandwidth and the controller is hyper-fast, or the demand of flow setup is low to moderate. However, chances are good that neither is met. On one hand, our analysis and measurement show that the *round trip time* (RTT)—the interval between the instance `packet-in` is sent out and the instance `packet-out` or `flow-mod` is received—can be considerably long: for instance, according to our measurement (Section IV), RTT is 3.67–4.06 ms even in low-load conditions, which is in line with what others (e.g., [4], [15]) reported. But on the other hand, the flow inter-arrival time (reciprocal of flow setup request rate) can reach 0.1–10 μs [5], [6], which is several orders shorter than the RTT. In general cases, and for a more rigorous understanding, we analyze the problem as below.

A. Analysis

Let us consider a network with s edge switches, each of which connects to h hosts. The data traffic between the hosts is specified by the following matrix:

$$\begin{array}{c} i\backslash j \\ 1 \\ 2 \\ \vdots \\ sh \end{array} \begin{array}{c} \begin{array}{cccccccc} 1 & 2 \ldots & h & h+1 \ldots & 2h & \ldots & sh \end{array} \\ \begin{pmatrix} 0 & \lambda_{12} \ldots & \lambda_{1h} & \lambda_{1,h+1} \ldots & \lambda_{1,2h} & \ldots & \lambda_{1,sh} \\ \lambda_{21} & 0 \ldots & \lambda_{2h} & \lambda_{2,h+1} \ldots & \lambda_{2,2h} & \ldots & \lambda_{2,sh} \\ \vdots & \vdots & & \cdots & & & \vdots \\ \lambda_{sh,1} & \lambda_{sh,2} & & \cdots & & & 0 \end{pmatrix} \end{array}$$

where λ_{ij} is the packet arrival rate at host i destined to host j, and $i, j = 1, 2, ..., sh$.

Denote by RTT_k the average round trip time for switch k where $k = 1, 2, ..., s$. Denote by N_k the average number of `packet-in` that switch k sends during RTT_k, i.e., N_k/RTT_k is flow setup request rate. Consider the *worst case*, where either no flow-table entries have been set up or all the entries have *expired* (OpenFlow supports both idle-entry timeouts and hard timeouts); in other words, switch k's flow table is empty. In this case, each packet arrival will trigger a `packet-in` to be sent and N_k achieves the maximum:

$$N_k^{max} = RTT_k \sum_{i=(k-1)h+1}^{kh} \Big(\sum_{j=1}^{(k-1)h} \lambda_{ij} + \sum_{j=kh+1}^{sh} \lambda_{ij} \Big) \tag{1}$$

where it is assumed that switch k is configured to be aware of the set of h hosts attached to it and will only consult the controller when the destination host is not attached to it. Otherwise, the switch will consult the controller for every destination host and Eqn. (1) becomes

$$N_k^{max} = RTT_k \sum_{i=(k-1)h+1}^{kh} \sum_{j=1}^{sh} \lambda_{ij} \tag{2}$$

which does not alter the problem in principle. Henceforth, we focus on Eqn. (1).

[1]An OpenFlow channel can be provisioned either in band or out of band, which is out of the scope of OpenFlow.

B. Numerical Results

To obtain concrete values, let us consider a three-level Clos network [16] that consists of $s = 80$ edge switches with 8 uplinks each, 80 aggregation switches, and 8 core switches, depicted in Fig. 2. Each edge switch is attached to $h = 20$ servers.

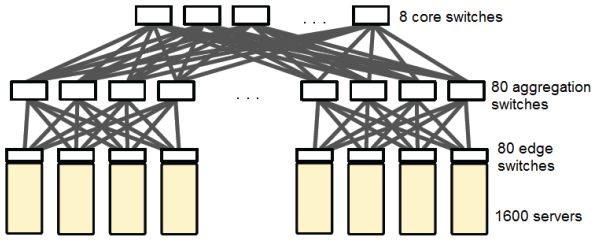


Fig. 2. A Clos network with 168 switches and 1600 servers.

In a typical data-center setting, each server transfers 128MB to every other server per second with 1500-byte packets, each link is of capacity 1 Gbps, OpenFlow channels are provisioned out of band with bandwidth 1 Gbps, whereby the controller can accomplish 275 flow setups per second [4]. This setting translates to a packet arrival rate of $\lambda_{ij} \approx 85.3$k packet/sec or inter-arrival time of 11.7μs, as well as $RTT_k = 1/(275/80) \approx 291$ ms. It then follows from Eqn. (1) that

$$N_k^{max} \approx 7.84 \times 10^8, \ k = 1, 2, ..., 80, \tag{3}$$

or 2.7 Giga `packet-in` messages per second, which far exceeds OpenFlow channel bandwidth (1Gbps) and will likely cause congestion.

C. Algorithm

We propose a Control-Message Quenching (CMQ) scheme to address this problem. In CMQ, a switch will send, during each RTT, only one `packet-in` message for each source-destination pair upon (multiple) table-misses. The rationale is that packets associated with each such pair will be treated as in the same flow and the corresponding `packet-out` or `flow-mod` messages will carry redundant information and thus are unnecessary.

To achieve this, each switch maintains a dynamically updated list, $\mathcal{L}$, of source-destination address[2] pairs $\langle s, d \rangle$. For each incoming packet that fails to match any flow-table entry, i.e., a table-miss occurs, the switch checks the packet against $\mathcal{L}$ and only sends a `packet-in` if the packet's $\langle s, d \rangle \notin \mathcal{L}$. Otherwise, it simply inserts the $\langle s, d \rangle$ in $\mathcal{L}$ and queues the packet. After receiving the controller response, i.e., `packet-out` or `flow-mod`, those backlogged packets that match the address information encapsulated in the response will be processed accordingly, as per the original OpenFlow.

[2]Depending on the type of the switch, the address could be an IP address, Ethernet address, MPLS label, or MAC address.

A new flow entry will also be installed in the flow table for the sake of subsequent packets. This process is formally described in Algorithm 1.

Algorithm 1 Control-Message Quenching (CMQ)

```
1: L := ∅
2: for each incoming event e do
3:   if e ="an incoming packet" then
4:     look up flow table to match the packet
5:     if matched then
6:       handle the packet as per the matched entry
7:     else
8:       look up L for the packet's source-
         destination address pair ⟨s, d⟩
9:       if found then
10:        enqueue packet //suppresses packet-in
11:      else
12:        send packet-in to controller
13:        enlist ⟨s, d⟩ → L
14:      end if
15:    end if
16:  else if e ="incoming packet-out or
       flow-mod" then
17:    dequeue and process matched packets as per
       instruction
18:    install a new entry in flow table
19:    de-list corresponding ⟨s, d⟩ from L
20:  end if
21: end for
```

At line 12, as per OpenFlow, the associated packet will be queued at the ingress port in ternary content addressable memory (TCAM) if the switch has; otherwise, the `packet-in` will encapsulate the entire packet. At lines 10 and 17, the queue is the same TCAM queue as above if the switch has; otherwise, the switch shall allocate such a queue in its OS user or kernel space.[3]

D. Analysis Revisited

Let us reconsider the network described in Section III-A but with CMQ now. Let RTT'_k denote the round trip time for switch k when using CMQ, and N'_k the average number of `packet-in` sent by switch k during RTT'_k. Still considering the worst case, we have

$$N_k'^{max} = \sum_{i=(k-1)h+1}^{kh} \Big(\sum_{j=1}^{(k-1)h} \min\{1, \lambda_{ij} \times RTT'_k\} + \sum_{j=kh+1}^{sh} \min\{1, \lambda_{ij} \times RTT'_k\} \Big). \quad (4)$$

Each summation term is capped by 1 because at most one `packet-in` will be sent for each source-destination pair of hosts in one RTT.

Revisiting the example Clos network given in Section III-B, we first note that $\lambda_{ij} \times RTT_k \approx 24.8k \gg 1$. Hence, it is plausible to assume that $\lambda_{ij} \times RTT'_k > 1$ because RTT'_k is unlikely to be more than 4 orders smaller than RTT_k (which was later verified by our experiments, too). It then follows from Eqn. (4) that

$$N_k'^{max} = 31.6 \times 10^3, \ k = 1, 2, ..., 80. \quad (5)$$

Compared to Eqn. (3), it indicates a remarkable reduction of control messages. For a comparison on the same time scale using per second, i.e., N_k^{max}/RTT_k versus $N_k'^{max}/RTT'_k$, it becomes $25632 \times RTT'_k/RTT_k$ folds reduction of flow setup requests per second, which is still substantial since RTT'_k is unlikely to be more than 4 orders smaller than RTT_k as mentioned above.

Our worst-case analysis gives an upper bound to the (average) performance improvement, in terms of reduction of flow setup requests that CMQ can achieve.[4] This shows significant potential; to evaluate the average performance, we conduct experiments described next.

IV. Performance Evaluation

To quantify the performance gain with CMQ, we conduct experiments using Mininet [17], which is a prototyping system widely adopted by institutions including Stanford, Princeton, Berkeley, Purdue, ICSI, UMass, NEC, NASA, and Deutsche Telekom Labs. One important feature of Mininet is that users can deploy exactly the same code and test scripts in a real production network.

We use probably the most popular OpenFlow controller, NOX [18], and a popular OpenFlow switch model, Open vSwitch [19]. We also use Wireshark [20], a network protocol analyzer, to verify the protocol workflow.

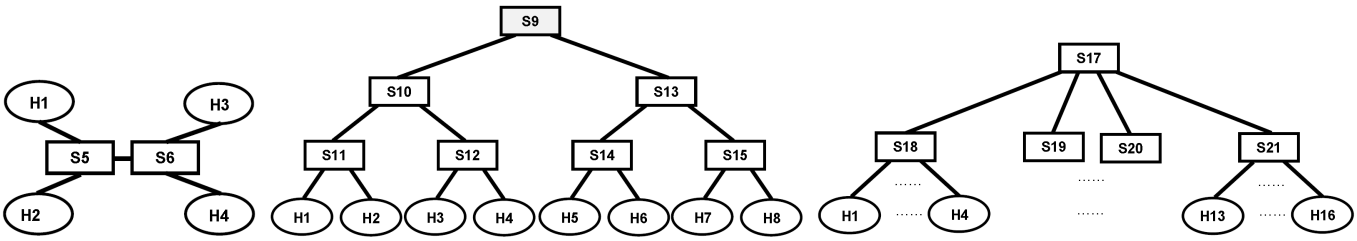
A. Experiment Setup

Our experiments use three topologies: Linear, Tree-8 and Tree-16,[5] as shown in Fig. 3. We employ two traffic modes, latency mode and throughput mode, as used by cbench [21]. In latency mode, a switch initiates only one outstanding flow setup request, waiting for a response from the controller before soliciting the next request. In throughput mode, each switch sends packets using its maximum rate, attempting to saturate the network.

[3] Although OS memory is generally slower than TCAM, it does not present a problem because the enqueueing process happens during RTT which is essentially an idle period; in the case of dequeueing, one can dedicate a thread for non-blocking processing.

[4] It is not obvious, though, that $N_k^{max}/N_k'^{max}$ gives an upper bound to average performance, while replacing $N_k'^{max}$ with N'_k would be more intuitive. In fact, because capped by 1 as in Eqn. (4), the difference between $N_k'^{max}$ and N'_k is by far smaller than that between N_k^{max} and N_k. Hence, what we derive is a *de facto* upper bound.

[5] Both tree and Clos topologies are typical in data center networks; we did not use Clos because Mininet does not allow loops to appear in its topology.



(a) Linear: 2 switches. (b) Tree-8: depth=3 and fanout=2. (c) Tree-16: depth=2 and fanout=4.

Fig. 3. Topologies used in experiments. Rectangles represent switches and ellipses represent hosts.

The traffic pattern is designated as follows. In latency mode, the first host sends packets to the last host and the others do not generate traffic. That is, H1→H4 in Linear (Fig. 3a), H1→H8 in Tree-8 (Fig. 3b), H1→H16 in Tree-16 (Fig. 3c). In throughput mode, in Linear, H1→H3 and H2→H4; in Tree-8, H_i sends packets to $H_{(i+4) \bmod 8}$, i.e, H1→H5, ..., H4→H8 (treating H8 as H0), H5→H1, ..., H8→H4 (8 flows); in Tree-16, H_i sends packets to $H_{(i+4) \bmod 16}$ (16 flows). Therefore, the hop count of each route in Tree-8 or Tree-16, is equal (6 and 4, respectively).

The metrics we evaluate are (1) RTT: flow setup latency as defined in Section III, which we measure on all switches and then take the average, and (2) end-to-end throughput, aggregated over all flows, which we measure using *iperf*. Note that RTT is essentially a control-plane metric and throughput a data-plane metric; using these two metrics allows us to see how the two planes quantitatively interact.

In all networks, link capacity is 1 Gbps, and OpenFlow channels are provisioned out of band with bandwidth also 1 Gbps. Flow-table entry timeout is 60 seconds by default. All the results are averaged over 10 equal-setting runs. In each run, the first 5 seconds are treated as warm-up and excluded from statistics.

B. Experimental Results

As a baseline, we first measure RTT for OpenFlow without CMQ and report the data in Table I. In throughput mode, RTT significantly increases by about 13, 143, 175 folds compared to latency mode on the three topologies, respectively. This clearly indicates congested OpenFlow channels and necessitates a remedy.

TABLE I
RTT IN OPENFLOW (UNIT: MS)

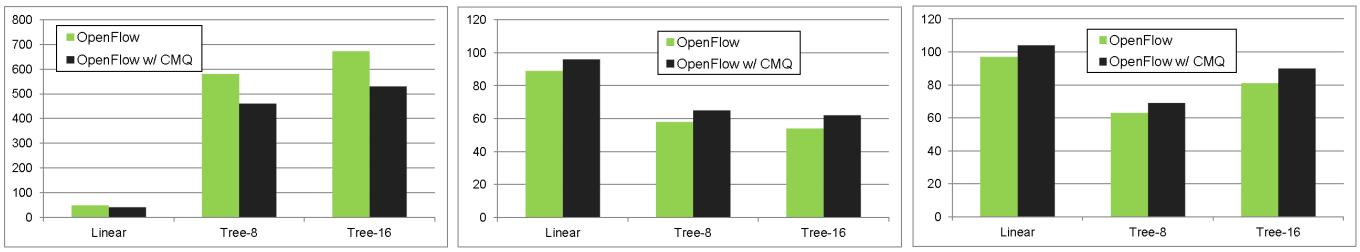
	Linear	Tree-8	Tree-16
Latency mode	3.67	4.06	3.83
Throughput mode	48.7	580.2	672.2

Next, we measure RTT for OpenFlow with CMQ in throughput mode (it should be understood that there will be no difference between OpenFlow with and without CMQ in latency mode) and present the results in Fig. 4a, where the RTT of original OpenFlow is reproduced from Table I. We can see that, CMQ reduces RTT by 14.6%, 20.7% and 21.2% in Linear, Tree-8 and Tree-16, respectively. This is considerable and, more desirably, the improvement increases when the network gets larger, which indicates an aggregating effect and is meaningful to network scalability.

To see how RTT affects throughput, we measure throughput in the same (throughput) mode, and show the results in Fig. 4b. We see that, with CMQ, network throughput is increased by 7.8%, 12.0% and 14.8% for Linear, Tree-8 and Tree-16, respectively. On one hand, this improvement is not as large as in the case of RTT, because what CMQ directly acts on is reducing RTT which only takes effect during flow setup and not during the lifetime of a flow. On the other hand, the improvement will become larger if the controller is configured in band, because the quenched control traffic will additionally alleviate traffic burden of the data plane as well.

Finally, we zoom in onto single-flow case by measuring the throughput for each network containing only one flow: H1→H4, H1→H8, H1→H16 in Linear, Tree-8 and Tree-16, respectively (as in the latency mode). The results are shown in Fig. 4c. In addition to observing the similar improvement to the multi-flow case as in Fig. 4b, we make two other observations from a cross-comparison with Fig. 4b: (1) For each topology, single-flow throughput is slightly higher than multi-flow throughput. This is because, in the multi-flow case, each topology effectively has at least one *shared* or bottleneck switch on all the routes, i.e, S5 and S6 in Linear, S9 in Tree-8, and S17 in Tree-16, which essentially prevents concurrency. (2) Tree-16 has lower multi-flow throughput but higher single-flow throughput than Tree-8. This is because in the single-flow case, H1→H16 in Tree-16 actually takes shorter path than H1→H8 in Tree-8 (4 hops versus 6 hops).

Remark: Our experiments demonstrate that CMQ considerably enhances controller responsiveness, evidenced by reduced RTT or shorter flow setup latency. As a direct result, this improvement in the control plane leads to increased throughput in the data plane. More importantly, it substantially benefits network scalability



(a) RTT (ms) in throughput mode. (b) Throughput (Mbps) in throughput mode. (c) Single-flow throughput (Mbps).

Fig. 4. Performance evaluation for RTT and throughput under different modes.

as the (central) controller is now much less involved in flow setup (most flow setup requests are suppressed by CMQ running on switches) and OpenFlow channels become more congestion-proof. This implies that more computational resource and bandwidth are conserved and network capacity is effectively expanded and can accommodate higher traffic demand.

V. Conclusion and Future Work

In this paper, we address OpenFlow scalability by enhancing controller responsiveness via a Control-Message Quenching (CMQ) scheme. Compared to prior studies that address the same issue, our solution requires minimal changes to OpenFlow switches, imposes zero overhead on the controller (which is bottleneck-prone), is lightweight and simple to implement. These features are important considerations in industry adoption.

Using analysis and experiments based on a widely-adopted prototyping system, we demonstrate that CMQ is effective in improving controller responsiveness which, ultimately, leads to more scalable OpenFlow networks.

This study also bolsters resource-constrained yet large-scale OpenFlow networks. For instance, it could be used by Sensor OpenFlow [22], the first work that synergizes OpenFlow and wireless sensor networks.

In future work, we plan to experiment with larger networks and more extensive settings including traffic patterns and parameters such as flow entry expiry time, and investigate the effect of in band channel provision.

References

[1] Open Networking Foundation, "Software-defined networking: The new norm for networks," white paper, April 2012.

[2] 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.*, vol. 38, no. 2, pp. 69–74, 2008.

[3] Open Networking Foundation, https://www.opennetworking.org.

[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 *ACM SIGCOMM*, 2011.

[5] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements and analysis," in *ACM Internet Measurement Conference (IMC)*, 2009.

[6] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *ACM Internet Measurement Conference (IMC)*, 2010.

[7] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: a distributed control platform for large-scale production networks," in *The 9th USENIX conference on Operating systems design and implementation (OSDI)*, 2010, pp. 1–6.

[8] A. Tootoonchian and Y. Ganjali, "HyperFlow: a distributed control plane for openflow," in *INM/WREN*. USENIX Association, 2010.

[9] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," in *ACM SIGCOMM*, 2010.

[10] A. S.-W. Tam, K. Xi, and H. J. Chao, "Use of devolved controllers in data center networks," in *IEEE INFOCOM Workshop on Cloud Computing*, 2011.

[11] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *IEEE INFOCOM*, 2011.

[12] Z. Cai, A. L. Cox, and T. S. E. Ng, "Maestro: A system for scalable openflow control," Rice University, Tech. Rep. TR10-08, 2010.

[13] A. Tootoonchian, S. Gorbunov, Y. Ganjali, and M. Casado, "On controller performance in software-defined networks," in *USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, 2012.

[14] Open Networking Foundation, "OpenFlow switch specification," April 16, 2012, version 1.3.0. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/specification/openflow-spec-v1.3.0.pdf

[15] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?" in *the 9th USENIX conference on Operating systems design and implementation (OSDI)*, 2010, pp. 1–6.

[16] C. Clos, "A study of non-blocking switching networks," *Bell System Technical Journal*, vol. 32, no. 2, 1953.

[17] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *ACM HotNets*, 2010.

[18] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, 2008.

[19] Open vSwitch, http://www.openvswitch.org.

[20] Wireshark, http://www.wireshark.org.

[21] R. Sherwood and K.-K. Yap, "Cbench: an openflow controller benchmarker," http://www.openflow.org/wk/index.php/oflops.

[22] T. Luo, H.-P. Tan, and T. Q. S. Quek, "Sensor OpenFlow: Enabling software-defined wireless sensor networks," *IEEE Communications Letters*, 2012, to appear.