Ryuo: Using High Level Northbound API for Control Messages in Software Defined Network

Shaoyu Zhang*, Yao Shen*, Matthias Herlich†, Kien Nguyen‡, Yusheng Ji†, Shigeki Yamada†
*Department of Computer Science and Engineering
Shanghai Jiao Tong University, China
Email: zsy@sjtu.edu.cn, yshen@cs.sjtu.edu.cn
†National Institute of Informatics, Japan
Email: herlich@upb.de, {kei, shigeki}@nii.ac.jp
‡National Institute of Information and Communications Technology, Japan
Email: kienng@nict.go.jp

***Abstract*—In the software defined networks (SDNs), the OpenFlow protocol is typically used as the southbound API in manipulating OpenFlow switches. However, the OpenFlow control messages are in a low abstraction level. Therefore, even a single application-level operation requires many OpenFlow messages, which consume the bandwidth of the control network and reduce the SDN's scalability. One potential solution is to use high level domain specific northbound APIs in the control network. In this paper, we explore the possibility of adopting this solution by implementing and evaluating a new SDN framework, Ryuo. In Ryuo, we introduce Local Service, which runs directly on each SDN switch (hardware/software). In operations, Local Service provides northbound APIs to the SDN applications while it can use different southbound APIs for different switches. Ryuo eliminates unnecessary control messages, hence it decreases the volume of control traffic. Our evaluation of Ryuo on Mininet with example applications shows that Ryuo reduces the volume of control traffic at least 50% compared to the standard OpenFlow, and up to 40% compared to the local controller approach. We also evaluate the performance of running Local Services directly on physical switches. The results show that we can achieve lower event handling latency in large networks, but with the trade-off of a lower event handling throughput due to the computing power limitation on physical switches. In summary, we have shown that using high level northbound API in the control network can make the control network more efficient, and leads to better scalability.**

I. Introduction

By separating the control plane and the data plane, Software Defined Network (SDN) gives us the opportunity to improve the programmability of networks. While the data plane is stable and focuses on forwarding, the control plane is implemented in software to gain flexibility. The southbound API is the interface implemented by SDN switches for the control plane. Most SDN switches available on the market implement the OpenFlow [1] southbound API. The OpenFlow southbound API provides a vendor independent interface for programming network devices. Scientists and engineers can experiment with different protocols and applications using OpenFlow controllers, instead of buying new devices or sending requests to vendors for new features. OpenFlow doesn't work well in all kinds of environments. It was initially designed for experimental use in small networks like campus networks and is still under active development. OpenFlow uses a centralized control model. Each switch must connect to a centralized controller via an out-band control network (a separate control network) or an in-band connection. All tasks that cannot be implemented with flow and group table entries must be done by the controller. The centralized control model of OpenFlow has problems such as low scalability [2] and high latency [3]. In OpenFlow's original model, all control tasks even the ones that only need local information have to be done by the controller, and those control messages have to travel through the control network. However, the bandwidth of the control network is often limited. An example of the standard setup of OpenFlow is shown in Figure 1. Southbound control messages are used in the control network in the standard setup of OpenFlow. The Northbound API is used locally in the Control Server. The Northbound API is the API implemented by SDN controllers and used by SDN applications. Some SDN controllers only provide a northbound API that is barely a simple wrapping of southbound API, but some SDN controllers such as OpenDaylight [4], provide high level APIs for different domains of applications [5].

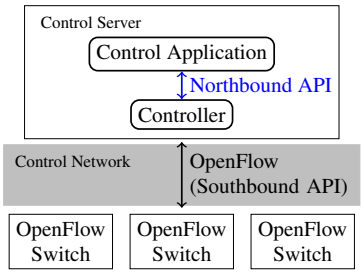


Fig. 1. The standard setup of OpenFlow. Southbound control messages are used in the control network, the northbound API is used on Control Server locally.

However, northbound control messages are much more expressive than southbound ones. If we use northbound control messages instead of southbound control messages in the control network, we can greatly reduce the control traffic. By reducing the control traffic, the resources in the control network can be used more efficiently. Also, if we use some southbound APIs in the control network, the functionality provided by the SDN application is limited by that specific southbound API. For example, we cannot use the fast failover functionality introduced in OpenFlow 1.1 when one of the switches only supports OpenFlow 1.0.

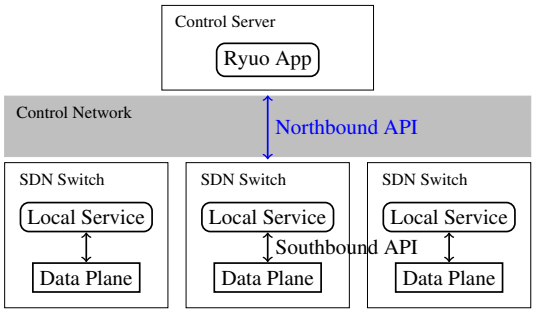


Fig. 2. The architecture of Ryuo, with northbound control messages in the control network.

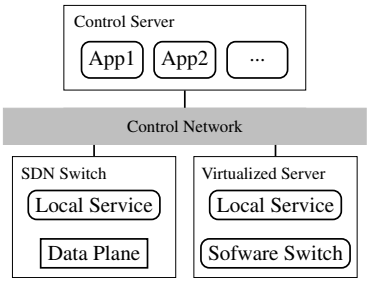


Fig. 3. Deployment of Ryuo. We can run Local Services directly on physical SDN switches no matter which southbound API is implemented, or on virtualized servers with software SDN switches like Open vSwitch. Ryuo applications are deployed like standard OpenFlow applications on a dedicated control server. Local Services and Ryuo applications can communicate through the control network.

In this paper, we explore the possibility of using high level northbound API for control messages in SDN. In Section III, we will discuss the design and implementation of our prototype framework with two example applications. In Section IV, we evaluate our framework on both Mininet and physical switches to explore the benefits and limitations of our approach. Finally, we show our conclusion in Section V.

II. Related Work

New SDN frameworks have been proposed to mitigate the drawbacks of the original OpenFlow model. Some proposals try to solve the scalability problem of OpenFlow. HyperFlow [6] introduces a distributed controller for OpenFlow. The system synchronizes the network-wide states between multiple controllers, by replaying events that affect controller states, which brings the overhead of event propagation, both on controllers and on the control network. Other distributed SDN controllers like ONOS [7] and OpenDaylight use distributed data store to maintain the consistency of network wide information. Onix [8] provides two kinds of distributed storage system, one provides strong consistency and one provides weaker consistency. It lets application developers decide the trade-off between performance and consistency. These work focus on the throughput of the control plane but leave the problem of propagation delay between control plane and data plane unsolved, which we solved in this paper.

One simple approach to maintain the consistency of network-wide information is not to distribute it. Instead, it is possible to offload some of the central controller's job to switches. DevoFlow [9] takes the path of augmenting the data plane, which can achieve good performance but will take some time to be adopted by switch vendors. Kandoo [10] uses local controllers running on physical switches or local servers to handle all local events. The local information is maintained only by each local controller. The root (global) controller subscribes to OpenFlow events from local controllers using Kandoo extension messages. Local controllers act like switch proxies. They receive and follow the OpenFlow control messages sent by the root controller, and propagate subscribed OpenFlow events to the root controller. By using local controllers, the latency of local events handling and the volume of control traffic can be decreased. However, by using northbound control messages in the control network, we can further reduce the control traffic, and also, we decouple SDN applications from southbound APIs. Orion [2] is a hierarchical control plane that focuses on the scalability of SDN applications. Each low level controller manages a portion of network and abstracts that portion as a node to the higher level controller. It reduces the computational complexity growth of the SDN control plane from super-linear to linear. Orion is a macro architecture for a network of networks, however, our work, like Kandoo, is focusing on the micro architecture for a single network.

Frenetic [11] is a programming language for SDN. Frenetic uses a run time environment to take care of all low level details of OpenFlow. Although the policy defined in the Frenetic language will be finally compiled into OpenFlow messages and sent through the control network, the bandwidth of the control network is still not used efficiently. Beehive [12] creates a general event-oriented programming model for SDN applications. Each OpenFlow application can be developed as a set of OpenFlow event handling functions. By analyzing the data dependency between these handling functions, Beehive exploits the opportunity of distributed processing. Based on the shared states between different handling functions, the system determines the execution location for each function. When locating a handling function, Beehive tries to make the executer close to the event generator. For an extreme example, if no state is shared between handling function F and all other handling functions, the function F can run directly on switches. Beehive is very flexible and solves both scalability and event handling latency. However, when using Beehive, application logic has to be separated into different low level OpenFlow event handling functions, which makes the application less readable and hard to maintain. In our proposal, only high level events are exposed to the central application, which avoid the fragmentation of the business logic.

III. Design and Implementation

To use northbound APIs for control messages, we need to put controllers on physical switches. Many OpenFlow switches are using Linux based operating systems, so it is possible to run custom programs on them. In virtualized environments where many virtual machines (VMs) running on the same physical server, operators often use Open vSwitch [13] to manage forwarding to each VM, so that we can run a controller on these physical servers.

With local controllers, we can run local applications on each physical switch. We need to split an entire SDN application into local and global parts. We let local applications

provide mechanism and maintain local information, while global applications provide policy and maintain network-wide information. There are different domains of SDN applications, such as routing, traffic engineering and access control. For each application domain, we can implement a local application, which we call *Local Service*. Each Local Service provides common programming interfaces, the domain specific northbound API, for its application domain. With Local Services, we free the northbound API from limitations given by southbound APIs. For example, we can implement Group Table functionality in a Local Service on older switches which only support OpenFlow 1.0. We can also make use of non-standard functionalities provided by vendors. For example, we can use BFD (Bidirectional Forwarding Detection) provided by Open vSwitch to detect a link failure with small latency, and we can implement BFD in a Local Service if some OpenFlow switches do not have built-in BFD support.

We call the global part of an application *Ryuo application*. For example, in centralized IP routing, different Ryuo applications may use different algorithms to compute routing entries, but they can use the same API to install routing entries. The deployment of Ryuo is shown in Figure 3. Ryuo applications running on a control server and Local Services either run directly on physical SDN switches or run on the same host with software SDN switches.

Local Services for an application domain can provide the same API even if they are running on switches that implement different southbound APIs. With the extra abstraction provided by Local Service, SDN applications can have a uniform view of network devices it manages. With the API provided by Local Services, we can focus on business logic instead of manipulating flow tables when developing SDN applications. Ryuo applications will not receive any notification of low level events such as *packet_in* in OpenFlow, but we still need notifications of high level events, such as *link up/down* for topology discovery and *access denied* for access control. For each application domain, Ryuo applications must provide several interfaces for Local Services to report high level events asynchronously. A topology discovery application must handle events like link up/down. A traffic engineering application may want to take action when a congestion is detected. In practice, only one Ryuo application in the same domain can be active in the network at the same time to avoid decision conflict. However, Ryuo applications can provide API for other Ryuo applications, so that applications from different domains can cooperate with each other. For example, some Ryuo application can simply query the topology discovery application for the network topology. Ryuo applications can specify applications they depend on. When the application starts, all required applications will also be loaded. Ryuo applications can also provide an administrative interface for the network operator.

Ryuo's architecture also has some limitations. First of all, the performance of Local Services is limited by the computing power on physical switches. However, SDN vendors begin to add more computing power to their new switches [14], so that this problem can be solved in the future. Second, since the code of Local Service resides on each switch, the complexity of deployment is increased. We can use automated deployment tools to mitigate this problem. The third problem is that Local Services on the same switch could make conflicting decisions. Simply assigning priority to each Local Service is not enough. Extra efforts must be taken to develop a mechanism to resolve these conflicts. We leave the decision conflict problem for future work.

To illustrate how Ryuo works and evaluate it later, we have developed two Local Services, the IP Routing Service and the Topology Service (we omit *Local* for names of Local Services hereafter for simplicity). With the IP Routing Service, the switch can behave like an IP router, IP routing applications can assign IP addresses to the switch and install routing entries. The Topology Service sending/receiving LLDP (Link Layer Discovery Protocol) [15] packets to/from peer switches reports the peer port on the neighbor switch to the topology discovery application when it finds a link is up, otherwise it reports that a link is down when the number of LLDP packets lost on the link exceeds a threshold.

We have also implemented two Ryuo applications: KeepForwarding Routing and Topology Discovery. The Topology Discovery application simply uses all the link information from Topology Services to maintain a global topology of the network. The KeepForwarding application uses the KeepForwarding [16] algorithm to compute routing entires based on the topology discovered by the Topology Discovery application. For each switch, the KeepForwarding algorithm can compute a list of output ports for every (*in_port*, *destination_IP*). All matched packets will be forwarded to the first available port in the output port list. In this way, the algorithm can achieve local link recovery without contacting the global control plane for rerouting. To maintain compatibility among different routing algorithms, our IP Routing Service can accept routing entries that match (*in_port*, *destination_IP*) and output to the first available port in a list of candidate ports. We can use fast failover group introduced in OpenFlow 1.1 to implement the candidate ports mechanism. But this feature is not in OpenFlow 1.0. We implement this particular algorithm as an example of decoupling the northbound API from the southbound API. We can use the framework in heterogeneous networks, where switches may implement different versions of OpenFlow or even different southbound APIs, the compatibility is guaranteed by Local Services. We also show the possibility of extending the switch with IP Routing Service.

Both Local Service and Ryuo application are implemented as Ryu [17] application. Ryu is an SDN framework which follows the original model of OpenFlow. The communication between Ryuo applications is implemented with Ryu events. The communication between Ryuo applications and Local Services is implemented with Pyro4, an open source remote procedure call library for Python.

Currently, the computing power on most SDN switches is poor when low cost CPUs are employed. We need to explore whether it is feasible to put Local Services on a physical switch. We will present our evaluation result and analysis in Section IV.

IV. Evaluation

We perform two kinds of evaluations.

- To evaluate how much control traffic can be reduced by the Ryuo framework, compared to standard Open-

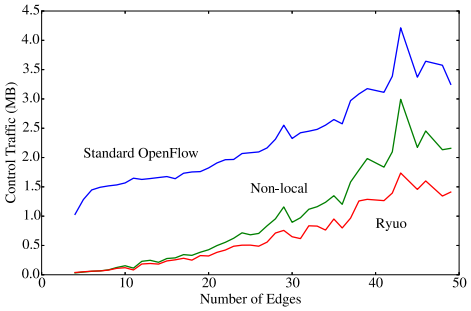


Fig. 4. Mean control traffic generated by the KeepForwarding application in the Ryuo and the standard OpenFlow setup on real world topologies with different number of edges. *Standard OpenFlow* represents the total control traffic generated by the standard OpenFlow setup. *Non-local* is the contol traffic generated by the standard OpenFlow setup that cannot be processed locally, we use this value to approximate the performance of local controller approaches. *Ryuo* is the total control traffic generated by Ryuo implementation.

Flow setup, and other approaches based on local controllers (in Subsection IV-A).

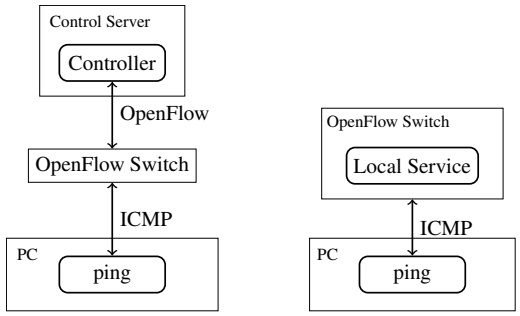
- To explore whether it is feasible to run Local Services directly on physical SDN switches which have limited computing power. The event handling latency and throughput evaluation for Local Services are discussed in Subsection IV-B and IV-C, both of them belong to this class.

In the throughput and latency evaluation, we use Pica8 P-3295 switches [18] with 825MHz PowerPC CPU and 512M memory. We use PCs with 3.2GHz duo-core, 4 threads CPU and 4G memory as control servers for standard OpenFlow applications and Ryuo applications. We use the Ryu framework in standard OpenFlow setups.

Since for the same SDN application, the generated control traffic only depends on the size of the network and the control messages used by the global control plane, we can perform the evaluation on Mininet [19] with Open vSwitch.

A. Control Traffic

One benefit we get from Ryuo is that Ryuo Applications communicate with Local Services in high level northbound control messages, which is more expressive than general southbound control messages like the one in OpenFlow. These high level interfaces not only improve productivity of application developers, but also decrease the amount of the control traffic. In this evaluation, we compare the amount of control traffic generated by the KeepForwarding Ryuo application and their counterparts implemented with a standard OpenFlow controller. In both setups, the topology application is loaded as a dependency of the routing application. We evaluate the application on real world topologies from the Topology Zoo Project [24]. We wait for the topology discovery application to get the topology of the emulated network. Then, we assign IP addresses to each IP Routing Service by sending HTTP requests to KeepForwarding application. HTTP requests are not included in control messages, since they are not in the control network. After the assignment of IP addresses, we use HTTP requests to request the computation and distribution of routing entries. Finally, we issue a *pingall* command to



(a) Standard OpenFlow setup with controller on PC.

(b) Local Service setup.

Fig. 5. System setups for comparing ICMP handling latency.

test all to all connectivity. This command also lets each IP Routing Service or the standard OpenFlow application handle ARP requests, and install flow entries to forward packets to end hosts. We record all traffic in the control network during the process. Since some of the topologies in the Topology Zoo may have the same number of edges, we use the mean results of all topologies with the same number of edges. We can obtain the *Non-local* results by ignoring all traffic that can be handled locally if we use local controller approaches like Kandoo. These local traffic includes control traffic that deals with LLDP, ARP handling. Because Kandoo's implementation is not publicly accessible, we use these results to estimate the performance difference. The portion of reduced control traffic in *Ryuo* compared to *Non-local* is contributed solely by using northbound control message, instead of using southbound control message, in the control network. The result is shown in Figure 4.

Ryuo can reduce control traffic by at least 50% compared to the standard OpenFlow setup, and up to 40% compared to local controller approach. For the smallest topology in our evaluation, the *Standard OpenFlow* result still shows a control traffic of about 1 MB. This is because we run the evaluation for 500 seconds for each topology, and in this time period, the standard OpenFlow topology discovery application sends and receives LLDP packets through *packet_in*/*packet_out* at a small time interval, which contributes the most control traffic in small topologies.

B. The Latency of Event Handling

Event handling latency is an important performance metric in SDN. If the delay between the data plane and the control plane is large, it would be infeasible to perform real time tasks and other tasks will have unacceptable performance degradation [20]. We evaluate the latency of two tasks in the IP Routing Service. The first one is ICMP Echo request handling. The second one is the implementation of fast failover group with OpenFlow 1.0 southbound API. The first one is a typical trivial task that can be done locally. The second one represents functionality extensions which we can implement in Local Services.

1) ICMP Handling: The evaluation setup is shown in Figure 5. A controller running on the control server communicates with the OpenFlow switch through the control network. The

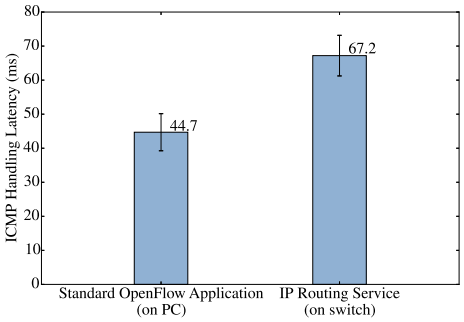


Fig. 6. ICMP handling latency comparison between a standard OpenFlow controller running on a control server and an IP Routing Service running on a Pica8 switch. The error bar represents the 95% confidence interval.

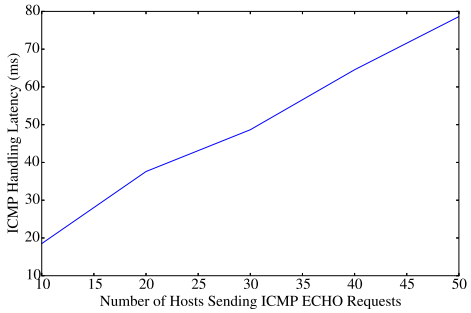


Fig. 7. In the standard OpenFlow setup, the central controller needs to handle all requests from switches in the network. The event latency grows with the number of requests.

latency of ICMP handling with standard OpenFlow setup includes an RTT (Round Trip Time) between control server and switch, and the processing delay in the controller. However, our evaluation network is very small and the propagation delay between the control server and the switch can be ignored.

To evaluate the performance of the IP Routing Service, we deploy the service on a Pica8 switch (see Figure 5b). In this evaluation, the global control plane is not involved. Therefore, we didn't put the Ryuo application in the figure. The IP Routing Service controls the switch through the local loopback device using OpenFlow. Note that the delay on the loopback device can be ignored.

In both setups, we ping the OpenFlow switch, since the IP Routing Service makes it behave like an IP router, the OpenFlow switch has its own IP address, and will handle ICMP requests sent to it. Because the delay of *packet_in* and *packet_out* is the same, in both cases we can ignore the delay between the ICMP handler and the data plane, the only difference is the processing delay of the ICMP handler. We can use the RTT of ping echo to estimate the latency of the ICMP handling. The results (the mean value of 100 RTTs collected) are shown in Figure 6.

In this evaluation, if we use a large network where the propagation delay between switches and the controller is larger than 10ms, IP Routing Service will outperform the standard setup for ICMP Echo request handling. Also, the control application has to handle events from more than one switch. We evaluate the relation between the event handling latency

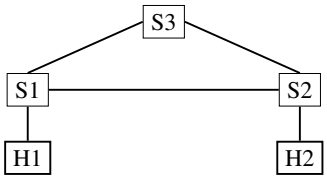


Fig. 8. Testing topology for the fast failover time evaluation. The normal route from $H1$ to $H2$ is $H1 \rightarrow S1 \rightarrow S2 \rightarrow H2$. After we tearing down link $S1 - S2$, the route will be $H1 \rightarrow S1 \rightarrow S3 \rightarrow S2 \rightarrow H2$.

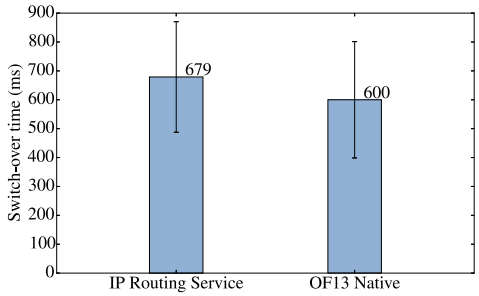


Fig. 9. Fast fail-over time comparison between the IP Routing Service implementation and the native implementation of the Pica8 switch. The error bar represents the 95% confidence interval.

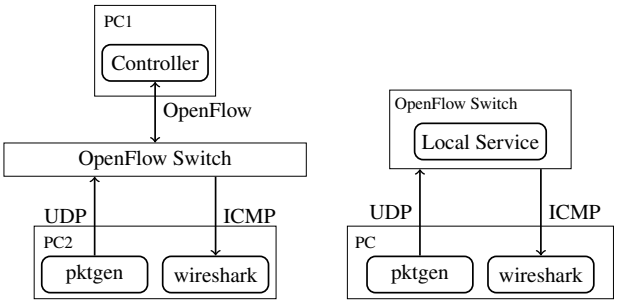
and the network size. We use Mininet to create networks with different number of switches, with the Routing Service deployed on each switch, and attach a host to each switch, then we let each host ping the switch attached to it. We collect 100 RTTs from each host. The result is shown in Figure 7.

2) Fast Failover Time: In this evaluation, we compare the fast failover time of the OpenFlow 1.3 fast failover group on a physical switch, and the IP Routing Service implementation running on a physical switch in OpenFlow 1.0 mode. The testing topology we use is shown in Figure 8. We use two PCs ($H1$ and $H2$) and three physical switches ($S1$, $S2$ and $S3$). We send packets with continuous sequence numbers from $H1$ using *pktgen* [21], and using *wireshark* [22] to capture all packets on $H2$. We disconnect the link $S1 - S2$ by unplugging the cable on $S2$ to let fast failover group entries on $S1$ take effects. We count the number of lost packets to analyze the fail-over time. A similar evaluation is also used in [23]. For the IP Routing Service implementation, the service keeps track of the status of all ports on the switch. When the status of a port changes, the service will install new flow entries for affected fast failover groups according to the status of switch ports.

The evaluation result is shown in Figure 9. The fail-over time of the IP Routing Service is less than 100ms slower when compared to native implementation of the Pica8 switch. Since we haven't optimized our implementation and the computing power is relatively poor, the performance could still be improved. With vendors adding more computing power to SDN switches, when the fast failover is implemented as switch extension in Local Service, the performance overhead could be smaller in the future.

C. The Throughput of Event Handling

We compare the event handling throughput of the IP Routing Service running on a physical switch and a standard



(a) Standard OpenFlow setup with controller on PC.

(b) Local Service setup.

Fig. 10. Setups for local event handling throughput comparison.

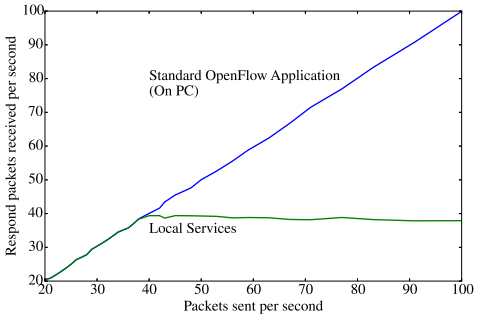


Fig. 11. Throughput evaluation result comparison between Ryuo Local Service running on a switch and an standard OpenFlow application running on a PC.

OpenFlow application running on a PC. Again, we use *pktgen* to send UDP packets to the switch at different rates, and the switch will respond with ICMP messages. For each packet rate, we send packets for 15 seconds, and use the response packets captured to compute the throughput. The standard OpenFlow setup is shown in Figure 10a and the IP Routing Service setup is shown in Figure 10b. The result is shown in Figure 11. We find the Local Service can only handle about 40 packets per second while we didn't reach the throughput limitation of the PC. The IP Routing Service performs poorly in this evaluation. However, in real networks, the central control application has to handle events from the entire network, we can expect a better scalability from Ryuo.

V. Conclusion

In this paper, we explored the potential benefits and limitations if we use high level northbound API for control messages in SDN. We developed a prototype framework called Ryuo. In Ryuo, we use Local Services to provide high level northbound APIs to global Ryuo applications. The Local Service decouples the northbound API from the southbound API, and reduces the control traffic volume. However, due to the poor computing power on current SDN switches, the performance of Local Services is still limited. This could be solved in the future by having more computing power on SDN switches.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[2] Y. Fu, J. Bi, K. Gao, Z. Chen, J. Wu, and B. Hao, "Orion: A hybrid hierarchical control plane of software-defined networking for large-scale networks," in *IEEE 22nd International Conference on Network Protocols (ICNP)*, 2014, pp. 569–576.

[3] K. Phemius and M. Bouet, "Openflow: Why latency does matter," in *IFIP/IEEE International Symposium on Integrated Network Management*, May 2013, pp. 680–683.

[4] The OpenDaylight Project. [Online]. Available: http://www.opendaylight.org/

[5] F. Schneider, T. Egawa, S. Schaller, S.-i. Hayano, M. Schöller, and F. Zdarsky, "Standardizations of sdn and its practical implementation," *NEC Technical Journal, Special Issue on SDN and Its Impact on Advanced ICT Systems*, vol. 8, no. 2, 2014.

[6] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *Proceedings of the Internet Network Management Conference on Research on Enterprise Networking*. USENIX, 2010, pp. 3–3.

[7] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "Onos: Towards an open, distributed sdn os," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. ACM, 2014, pp. 1–6.

[8] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A distributed control platform for large-scale production networks." in *OSDI*, vol. 10, 2010, pp. 1–6.

[9] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 254–265.

[10] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. ACM, 2012, pp. 19–24.

[11] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *SIGPLAN Notices*, vol. 46, no. 9. ACM, 2011, pp. 279–291.

[12] S. Hassas Yeganeh and Y. Ganjali, "Beehive: Towards a simple abstraction for scalable software-defined networking," in *Proceedings of the 13th Workshop on Hot Topics in Networks*. ACM, 2014, pp. 13:1–13:7.

[13] Open vSwitch. [Online]. Available: http://openvswitch.org/

[14] AS5712-54X with ONIE / 10GbE Data Center Switch. [Online]. Available: http://www.edge-core.com/ProdDtl.asp?sno=457&AS5712-54X

[15] "IEEE Standard for Local and metropolitan area networks - Station and Media Access Control Connectivity Discovery," *IEEE Std.*, 2005.

[16] B. Yang, J. Liu, S. Shenker, J. Li, and K. Zheng, "Keep forwarding: Towards k-link failure resilient routing," in *INFOCOM, 2014 Proceedings IEEE*, April 2014, pp. 1617–1625.

[17] Ryu SDN Framework. [Online]. Available: http://osrg.github.io/ryu/

[18] Pica8 P-3295 Switch. [Online]. Available: http://www.pica8.org/documents/pica8-datasheet-48x1gbe-p3290-p3295.pdf

[19] Mininet: An Instant Virtual Network on your Laptop (or other PC). [Online]. Available: http://mininet.org/

[20] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. ACM, 2012, pp. 7–12.

[21] R. Olsson, "Pktgen the linux packet generator," in *Proceedings of the Linux Symposium*, vol. 2, 2005, pp. 11–24.

[22] wireshark. [Online]. Available: https://www.wireshark.org/

[23] N. L. Van Adrichem, B. J. Van Asten, and F. A. Kuipers, "Fast recovery in software-defined networks," in *Third European Workshop on Software Defined Networking (EWSDN)*, 2014.

[24] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *Selected Areas in Communications, IEEE Journal on*, vol. 29, no. 9, pp. 1765 –1775, October 2011.