Support Multiple Auxiliary TCP/UDP Connections in SDN Simulations Based on ns-3

Hemin Yang, Chuanji Zhang, George Riley
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia
{hyang350,jenny_zhang,riley}@gatech.edu

ABSTRACT

As software-defined networking (SDN) grows beyond its original aim to simply separate the control and data network planes, it becomes useful both financially and analytically to provide adequate mechanisms for simulating this new paradigm. A number of simulation/emulation tools for modeling SDN, such as Mininet and ns-3, are already available. However, none of them supports multiple auxiliary connections in SDN simulation/emulation, which is one of the key features added in OpenFlow 1.3. In this paper, we extended the existing OFSWITCH13 framework to support multiple auxiliary connections in SDN simulations based on ns-3. This work allows multiple TCP/UDP connections to be built between a switch and a controller in the simulation. Furthermore, we performed case studies to investigate the impact of auxiliary connections on SDN performance, which is valuable for designing a scalable SDN network.

CCS CONCEPTS

•**Networks → Network simulations;** *Programmable networks;* •**Computing methodologies → Simulation tools; Simulation evaluation;**

KEYWORDS

Network Simulation; Software Defined Networking; Auxiliary Connection; ns-3

ACM Reference format:
Hemin Yang, Chuanji Zhang, George Riley. 2017. Support Multiple Auxiliary TCP/UDP Connections in SDN Simulations Based on ns-3. In *Proceedings of the 2017 Workshop on ns-3, Porto, Portugal, June 2017 (WNS3 2017),* 7 pages.
DOI: http://dx.doi.org/10.1145/3067665.3067670

1 INTRODUCTION

Software Defined networking (SDN) is an emerging networking technology, which can enable faster introduction of network innovations and radically simplify and automate the management of large networks. With newer and farther reaching applications being developed on SDN, it can prove beneficial both analytically and financially to employ modeling and simulation efforts toward initial developmental testing of its capabilities.

However, as far as we know, auxiliary connections, one of the key features added in OpenFlow 1.3 [6], is not supported by the existing simulation/emulation tools. *Mininet*, which is *de facto* standard emulator for SDN researches, can only build one connection between the switch and the controller by calling `start(controllers)` for each switch. OFSWITCH13 is a new module which enables ns-3 simulator to support OpenFlow 1.3 [3]. However, it does not support auxiliary connections either. The Direct Code Execution (DCE) approach in ns-3 can allow real and deployable SDN controller applications to run within ns-3 simulations [12]. Unfortunately, auxiliary connections can not be supported yet in this approach. The other simulators such as flow simulator fs-sdn [10] and OpenNet [2] can only support OpenFlow 1.0. On the other hand, many commercial/open source SDN controllers/switches support auxiliary connections, such as PicOS [11], FloodLight [9], OpenDayLight [4], and LINC [8]. For example, the OpenFlow plugin in OpenDayLight enables the setup for multiple auxiliary connections. Furthermore, the auxiliary connection is used to improve SDN performance in the academic community. A. Basta et al. [1] mentioned that the auxiliary connection is beneficial for control path migration for distributed SDN hypervisors. The auxiliary connection can also play a significant role in restoring network failure when a failure occurs in an SDN network [16]. Obviously, without a simulator/emulator supporting auxiliary connections, it is difficult to exploit the potential of auxiliary connections in improving the switch processing performance before the real-world deployment of SDN networks. For example, Z. Li et al. [14] proposed an OpenFlow channel deployment algorithm which optimizes the deployment of auxiliary connections. It is difficult, if not impossible, to verify the performance of this algorithm in a "real" network in the case where the simulator/emulator does not support auxiliary connections. Moreover, supporting auxiliary connections will allow researchers to examine the various factors influencing the load of the control channel and thus help develop to better load balance algorithms/mechanisms.

In this paper, we extended OFSWITCH13 to make it support multiple TCP/UDP auxiliary connections in ns3-sdn simulations. Furthermore, we presented our experiment results studying the performance of auxiliary connections with respect to the number of auxiliary connections, connection type, and network load in two networks. Instructions for downloading, installing, and using this extension can be found at https://github.com/meiwenPKU/OFSwitch13-Aux.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WNS3 2017, June 2017, Porto, Portugal
© 2017 ACM. 978-1-4503-5219-2/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3067665.3067670

2 AUXILIARY CONNECTIONS

Auxiliary connections are created along with the main connection, which are together served as the OpenFlow Channel between the switch and the controller. Different from the main connection, the auxiliary connection can be either unreliable (e.g., UDP, DTLS) or reliable (e.g., TCP, TLS). These auxiliary connections are helpful to "improve the switch processing performance and exploit the parallelism of most switch implementations" [6]. To support auxiliary connections in simulations, we need to be very clear how auxiliary connections are bound to the main connection, how auxiliary connections are setup, and what the constraints for packet delivery on auxiliary connections are.

2.1 Binding

Every connection is uniquely identified by its *Datapath ID* and *Auxiliary ID*, and the auxiliary connections should have the same *Datapath ID* as the main connection but different *Auxiliary ID*. Moreover, the auxiliary connections should have the same source IP addresses, destination IP addresses, and transport destination port number as the main connection. These constraints are used by the controller to bind the auxiliary connections to their main connection. In other words, once the controller receives the initialization of one connection, it should recognize whether it is an auxiliary connection or not. If it is, the controller should automatically bind it to the main connection with the same *Datapath ID*.

2.2 Setup and Maintenance

Auxiliary connections depend on the main connection. Only after the main connection is setup, the auxiliary connections can be initialized. On the other way around, the auxiliary connections should be closed immediately once the main connection is down. The connection setup for the auxiliary connections is the same as for the main connection. That is to exchange `OFPT_HELLO` messages to negotiate a common version number. If it is successful, the controller should then send a `OFPT_FEATURES_REQUEST` message to the switch to get the *Datapath ID* of the switch. However, the situation for the unreliable auxiliary connections is more complicated because the message order can not be kept. It is possible that the device receives other messages on the unreliable auxiliary connections before receiving a `OFPT_HELLO` message. In this case, the device should either assume that the connection is properly setup and use the version number from those messages or return an Error message. Once an OpenFlow device receives this error message, it should either send a new Hello message or close the unreliable auxiliary connection.

As for connection maintenance, it is very different from the main connection for unreliable auxiliary connections. The underlying TLS/TCP mechanisms of the main connection/reliable auxiliary connections can help maintain the connections. For the unreliable connections, the OpenFlow device needs a new mechanism to determine whether to terminate a connection. It should monitor the time interval between the receiving packets on an unreliable connection, and terminate the connection/send a new message if the interval is greater than some chosen threshold. If an auxiliary connection is terminated or broken, this should not impact the main connection or other auxiliary connections.

2.3 Message Delivery

In principle, all OpenFlow message types and sub-types can be transmitted on all connections. However, to provide QoS services, the auxiliary connections can be configured with different priorities. The switch and the controller will always first process the messages received on the auxiliary connection with high priority. In addition, both the controller and the switch can use the various connections for sending OpenFlow messages as they wish except for:

- A reply to an OpenFlow request must be made on the same connection the request came in;
- If messages must be processed in sequence, they must be sent over the same connection which does not reorder packets, and use barrier messages;
- The unreliable auxiliary connections (e.g., UDP) can only deliver messages such as `OFPT_HELLO`, `OFPT_PACKET_IN`, and `OFPT_PACKET_OUT`.

3 IMPLEMENTATION

3.1 OFSwitch13

The OFSwitch13 module enhances ns-3 with OpenFlow 1.3 technology support. The module components include the switch network device, the controller application interface, the OpenFlow channel, and the external *ofsoftswitch13* library.

The switch network device is used to interconnect ns-3 nodes using CSMA devices and channels. Each switch device consists of a collection of ports and each of the ports is connected to a CSMA device. The switch is connected to a controller either through a CSMA channel shared by all controlled switches or its own dedicated CSMA or point-to-point channel. Each switch receives packets from one port, directs them to the *ofsoftswitch13* library [7] for OpenFlow pipeline processing, and then executes the appropriate actions based on the action set collected from *ofsoftswitch*. An OpenFlow 1.3 controller interface and an OpenFlow channel also accompany the module to provide basic functionality for controller implementation. The *ofsoftswitch13* library provides the OpenFlow datapath implementation for OFSwitch13, including the input/output ports and flow, group, and meter tables. It uses the OFLib library to convert internal messages to and from OpenFlow 1.3 format. The NetBee library is used to decode and parse incoming packets. The library is modified to integrate with the OFSwitch13 module. In order to send and receive packets to and from the ns-3 environment directly, the related functions are annotated as weak symbols to permit overriding them at link time. A similar strategy is applied for time-related functions to ensure time consistency between the library and simulator. The library uses callbacks to notify the module about internal packet events.

3.2 Extension

To make OFSwitch13 support multiple auxiliary TCP/UDP connections, we extended all components in OFSWITCH13 except for the OpenFlow Channel, as shown in Figure 1. Different from the OpenFlow 1.3 specification, we make the controller to bind the

auxiliary connections with their main connections when the auxiliary connections are built in our implementation. In other words, the controller has the full knowledge of the main connections and their auxiliary connections in the simulated network before the simulation starts. Therefore, we allow the main connection and its auxiliary connections to own different IP addresses. This is convenient for ns-3 simulations because the built-in IP address assignment helper in ns-3 can make life easier.

As for the setup of an auxiliary connection, the switch device first calls `StartControllerAuxConnection` to create a TCP/UDP socket for the auxiliary connection. The socket will try to connect the remote controller. At the same time, the switch starts the `NoMsgAfterStartTimeout` timer, and closes the connection if no message was ever received on this connection before the timer expires. On the controller side, the controller will reject any TCP connection request from the auxiliary connections if the main connection is not setup. In addition, the controller starts to listen on the UDP auxiliary connections after the main connection is built (i.e., receiving `OFPT_HELLO` from the switch). In this way, an auxiliary connection can only be initialized when the main connection is setup. In addition, if the switch closes a connection gracefully or a connection is closed abnormally, the controller will first identify whether this connection is a main connection. If it is, the controller will close all auxiliary connections bound with the main connection.

To maintain an UDP auxiliary connection, several mechanisms are implemented in our extension. First, whenever the controller sends an `OFPT_FEATURE_REQUEST` to the switch, an event which checks whether the corresponding `OFPT_FEATURE_REPLY` is received on the same connection is scheduled to expire when the specific time `NoFeatureReplyTimeout` is reached. The controller will send a new `OFPT_FEATURE_REQUEST` message to the switch, if no reply is received before the timer is out. Moreover, another timer `NoMsgAfterRecTimeout` is started whenever a device (either controller or switch) receives a message on an UDP auxiliary connection. If no more message is received by the device on the connection before this timer is out, the UDP auxiliary connection is terminated. Another issue is how to deal with message reordering on an UDP auxiliary connection. If a device receives another message on an UDP auxiliary connection prior to receiving an `OFPT_HELLO` message, the device will return an `OFPT_Error` message with `OFPET_BAD_REQUEST` type and `OFPBRC_BAD_VERSION` code. Correspondingly, if the peer receives this error message, it will terminate the unreliable connection.

We also implemented a simple message scheduler for the switch and the controller based on OpenFlow 1.3, which is responsible for spreading packets across the various connections. This scheduler is subject to the packet delivery constraints discussed in Section 2.3, as well as:

(1) All Packet-Out messages containing a packet from a Packet-In message should be sent on the connection where the Packet-In came from;
(2) All other Packet-Out messages should be spread across the various auxiliary connections;
(3) All Packet-In messages are spread across the various auxiliary connections;
(4) All the other messages are sent over the main connection, if all the other constraints are satisfied.

Note that readers can develop their own schedulers, and they can modify `OFSwitch13Device::SendToControllerWithAux` and `OFSwitch13Controller::ReceiveFromSwitch` to deploy the scheduling policies.

In addition, OFSWITCH13 is based on the *ofsoftswitch13* library, which only supports one auxiliary connection. Therefore, we have to modify this library to provision multiple auxiliary connections. The main modification is in the *datapath.c*, where all the functions and structures related to the original auxiliary connection identifier `PTIN_CONNECTION` are modified to accommodate multiple auxiliary connections.

Finally, we also extend `OFSwitch13Helper` to provide API to help install the auxiliary connections between the controller and the switch. Instead of using `InstallSwitch` provided by OFSWITCH13, the users can just use `InstallSwitchWithAux` and pass a vector of `SocketType` as its parameter. This vector specifies the connection type (i.e., TCP and UDP) of each auxiliary connection. Moreover, `OFSwitch13Helper` also provides API to set the data rate and channel type (i.e., CSMA channel shared by all controlled switches, dedicated CSMA and point-to-point channel).

3.3 Stop the Simulation

In discrete event simulators, the simulation is stopped when the scheduled event list is empty. For SDN simulation, some underlying procedures will generate events periodically or irregularly forever. For example, the switch has to check whether the flow entry is time out periodically. Therefore, the only way to stop the simulation is to set the specific stop time. In ns-3, we can call `Simulator::Stop()` to terminate the simulation. However, in many cases, we do not know the exact time when the simulation should be stopped. For example, we fix the total number of bytes to send for each application installed on the host, and the simulation is supposed to stop when all these bytes are sent and the devices cannot receive more bytes. To support this kind of stopping criteria, we extended `PacketSink` application to stop the simulation when no packet is received during a chosen interval `IntervalSet`, which is an attribute of `PacketSink` and can be set by `PacketSinkHelper`. Specifically, whenever a packet is received, a global static variable `g_lastRevTime` will record the current time and a new event is scheduled to check whether *g_lastRevTime + IntervalSet* ⩽ *Simulator::Now()* holds at the relative time `IntervalSet` is reached. At the same time, the previous event scheduled in this way will be canceled because a new packet is received.

4 CASE STUDY

Experiments have been performed on the linear network and the campus network to show how to include multiple auxiliary TCP/UDP connections in the simulations and demonstrate the potential of the auxiliary connection in improving the performance of SDN network. We use `OFSWITCH13LearningController` provided by OFSWITCH13 in the simulations, which instructs the switches to forward incoming unicast frames from one port to the single correct output port whenever possible.

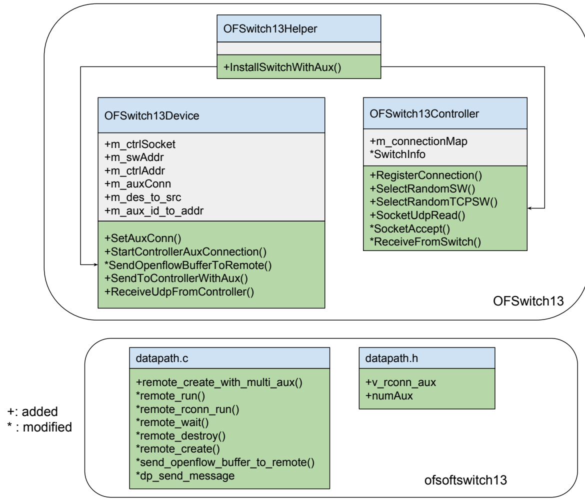

Figure 1: The extension of OFSWITCH13

As for traffic generation, the built-in `OnOffApplication` is used for traffic generation. This application can send to a specific IP address and port a maximum number of bytes (5120B) using a given packet length (512B). These packets can be configured to send through a TCP connection. By design, the `OnOffApplication` can be configured to generate traffic periodically rather than constantly. However, this capability is not used for these experiments. To confirm reception, the built-in `PacketSink` application with the extension described in Section 3.3 can be configured to listen on the same port.

Furthermore, in this case study, we want to show the impact of the auxiliary connection when the main connection is congested. Although the probability of a packet to be forwarded to the controller is reported to be 0.04 [13], the main connection is easily congested in practice in the case of large network topology (e.g., thousands of nodes) and low bandwidth of the main connection. According to A. R. Curtis et al. [5], the maximum bandwidth for flow-setup payloads between the switch (HP ProCurve 5406zl) and the controller is around 17 Mbps. We tested that it needs several thousand nodes to saturate the control channel with bandwidth 20 Mbps. On one hand, it demonstrates that the main connection can be congested (e.g., data centers are required to scale up to hundreds of thousands of nodes). On the other hand, with such a large topology, it takes several hours to finish the simulation. And we need to test 44 configurations in total, each of which is tested 10 times. Therefore, the conservatively estimated time for finishing all these simulations is about one month. In addition, we use `FlowMonitor` module in ns-3 to measure the delay of the received packets. This module uses probes, installed in network nodes, to track the packets exchanged by the nodes, which consumes a considerable amount of memory. Therefore, it is unfeasible to run the simulation for a network with thousands of nodes and flow monitor probes in our machine (an 8-core 3.6GHz Intel Xeon E5-1620 processor with 8GB of memory and running Ubuntu 14.04). Considering both time and memory cost, we scale down link bandwidth to fit the restricted resource budget [15]. In our simulations, we set the control channel bandwidth as 1 Mbps. The bandwidths for other links are also scaled down, which are specified in the following two subsections.

4.1 Linear Network

The linear network topology, as shown in Figure 2, is designed as a linear network of switches, where 10 access switches are connected one by one as a chain. Each switch also connects to a `OFSWITCH13LearningController` controller. In addition, every switch is connected to N hosts, where N is varied as 40, 60, and 80. For traffic generation, each host will randomly select another access switch and then send its data to its corresponding host on that switch, i.e. host 2 on switch 0 might send to host 2 on switch 8.

Network connections are all wired links with the following characteristics:

- Host to Access Switch: 1Mbps, 1ms delay
- Access to Access Switch: 10Mbps, 1ms delay
- Controller to Access Switch: 1Mbps, 1ms delay

4.2 Campus Network

The campus network topology examines a ring of simplified campus networks as shown in Figure 3. Each campus network is simply a ring of switches and each switch connects to its own set of hosts. In this work, 8 switches in the ring are considered as *access* switches, where each switch connects to a set of N ($N = 20, 30, 50$) hosts. An additional switch in the ring is considered as a *gateway* switch that connects to a single *exchange* switch. The exchange switches then form a ring themselves to connect all of the campus networks. The number of campus networks is fixed as 4. Each ring, including the exchange ring, is controlled independently by an `OFSWITCH13LearningController` controller. However, the spanning tree protocol is not implemented in this controller. To accommodate this limitation, one link from each ring in the network is deleted. In these simple ring topologies, this modification produces an effect similar to flooding loop prevention while maintaining roughly the same network behavior. The controller applications are all still allowed to execute as well to ensure as much network traffic is maintained as possible. For host traffic generation, each host will decide whether to send data to a remote ring with probability 0.1. Then, it will randomly select the switch and host to which it will send data.

Network connections are all wired links with the following characteristics:

- Host to Access Switch: 1Mbps, 1ms delay
- Access to Access Switch: 1Mbps, 1ms delay
- Access to Gateway Switch: 1Mbps, 1ms delay
- Access Switch to Controller: 1Mbps, 1ms delay
- Gateway to Exchange Switch: 10Mbps, 1ms delay
- Exchange to Exchange Switch: 10Mbps, 5ms delay
- Exchange Switch to Controller: 1Mbps, 1ms delay

4.3 Results and Discussion

In this work, we studied the performance of the auxiliary connections with respect to various network loads, connection types, and the number of connections. To measure the performance, we select flow setup delay, packet loss rate, and received packet delay

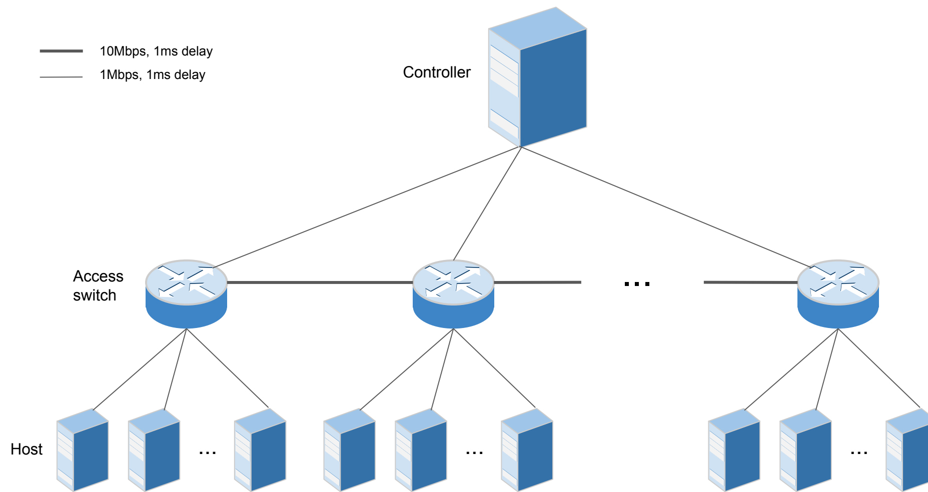

Figure 2: The linear network topology is a linear network of 10 access switches, each of which is connected to 40, 60, and 80 hosts. All switches connect to a single controller. This figure is regenerated from [12].

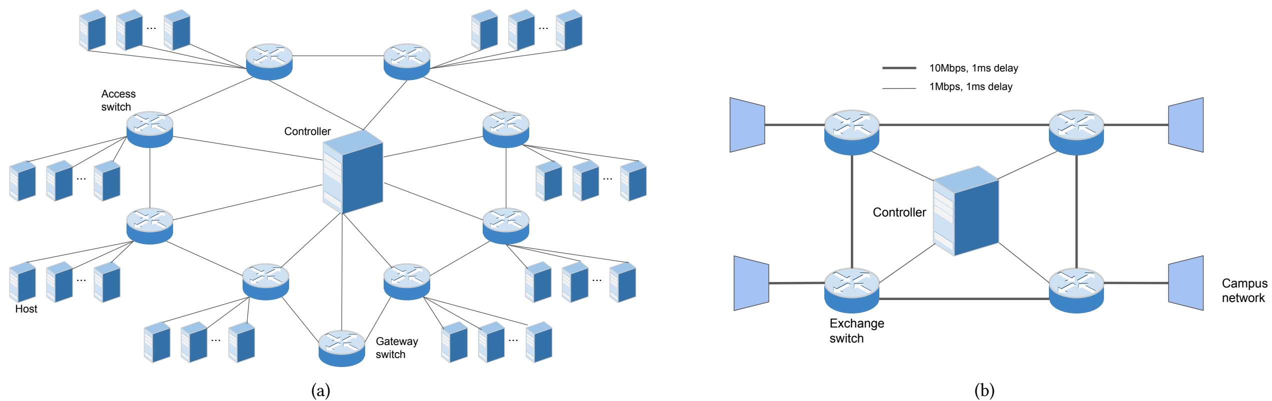

Figure 3: The campus network topology connects a ring of switches modeling a simplified campus network with identical rings, forming a ring of these smaller networks. Each campus network is composed of a ring of 8 access switches and a single gateway switch, where each access switch is connected to 20, 30, and 50 hosts. Each gateway switch connects to an exchange switch which is connected to other exchange switches, forming a larger ring. Each set of switches connects to its own controller. This figure is regenerated from [12].

as the performance metrics. Flow setup delay of a switch is the interval between sending a Packet-In message to the controller and receiving the corresponding Packet-Out message from the controller. (For the `OFSWITCH13LearningController` controller, a Packet-Out message with the same buffer id as the received Packet-In message is always sent back to the switch.) In addition, since we set the total number of bytes sent by each host as fixed in our simulations, packet loss rate can directly reflect network throughput (i.e., throughput = number of hosts in the network * number of bytes sent per host * (1-packet loss rate) / simulation time). Therefore, we do not use network throughput as one of the performance metrics.

As for received packet delay, as the name implies, it does not count the delay of the dropped/lost packets.

Results for the linear, and campus network topologies are shown in Figure 4 and Figure 5, respectively. For the simple linear switch topology, performance in terms of flow setup delay can be greatly reduced, even by three orders of magnitude. For example, the flow setup delay of the linear network with 80 hosts per switch can be reduced from 3s to 5ms when the number of TCP auxiliary connections is increased from 0 to 5. Auxiliary connections can also help reduce packet loss rate. We can see that the packet loss rate of the linear network with 80 hosts per switch is reduced from 49% to 15% when 5 TCP auxiliary connections are added. However, in the case

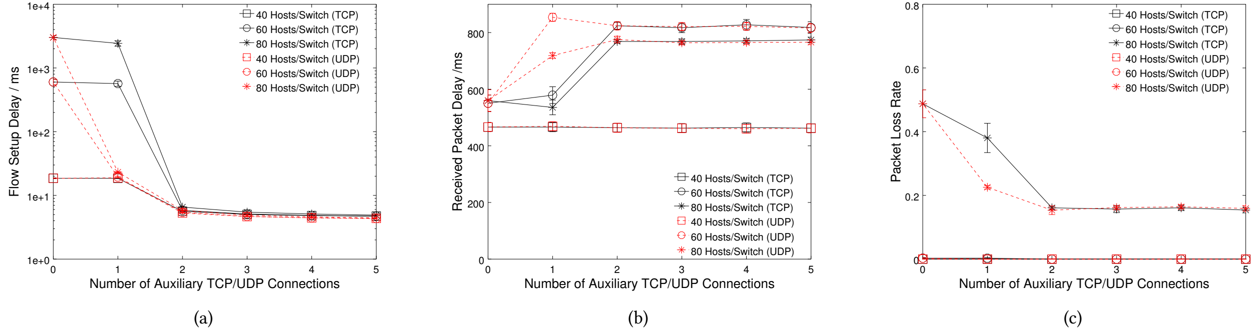

Figure 4: Linear network topology results for (a) flow setup delay in milliseconds, (b) received packet delay in milliseconds, and (c) packet loss percentage. The topology is a linear set of switches that each connect to a set of 40, 60, and 80 hosts per switch. A single controller manages and maintains layer 2 learning forwarding rules for the switches. Standard error bars are displayed for each data point and noted as negligible.

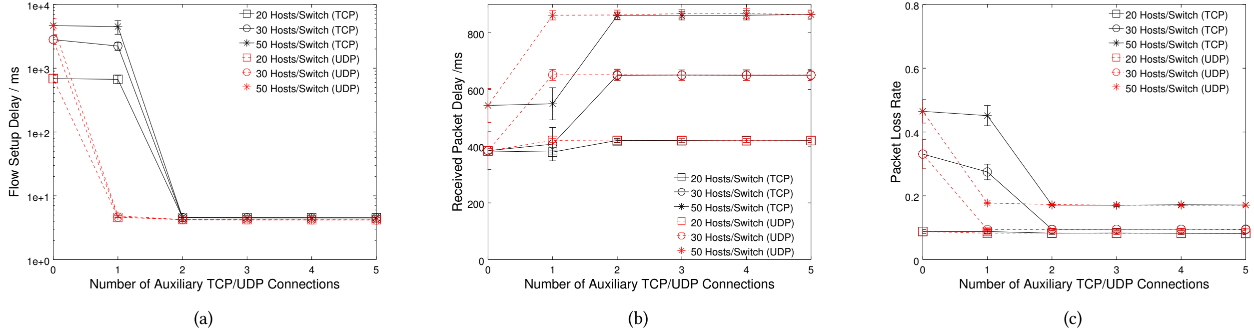

Figure 5: Campus network topology results for (a) flow setup delay in milliseconds, (b) received packet delay in milliseconds, and (c) packet loss percentage. Each campus network is a ring with one gateway switch connecting to the exchange ring of switches and 8 access switches that each connect to a set of 20, 30, and 50 hosts per switch. Each campus network as well as the exchange ring has its own controller. Standard error bars are displayed for each data point and noted as negligible.

of low network load, adding auxiliary connections has no impact on packet loss rate. For example, there is little performance gain when the number of TCP auxiliary connections is increased in the case of $N = 40$. This makes sense because the auxiliary connections are beneficial when the main connection is congested due to the massive packets deliveries on it. This result can be expected since the main control channel is not congested if there are fewer flows generated in the network. In terms of received packet delay, we can see that adding auxiliary connections can result in a larger delay. This makes sense because some of the packets supposed to be lost or dropped in the case of none auxiliary connection can be received with the help of auxiliary connections. These received packets always incur a large delay, which will increase the overall packet delay. Interestingly, we can find that adding one UDP auxiliary connection can achieve much higher performance gain comparing with TCP. Since UDP is a connectionless protocol, much of its performance gain is most likely a consequence of lightweight overhead (e.g., no ordering of messages, no flow control, no error recovery, etc.) of UDP. However, when the number of connections is greater than 3, both TCP and UDP achieve similar performance. Combining all these three metrics, we can see that, when the number of auxiliary connections reaches 3, adding more auxiliary connections does not help improve the performance. For the campus network topology, we can have similar observations including:

(1) Adding auxiliary connections can help reduce flow setup delay and packet loss rate greatly, at the cost of increasing the delay of the received packets;
(2) Adding more auxiliary connections can not improve SDN performance when there have already been 2 auxiliary connections;
(3) Adding one UDP auxiliary connection to the main connection has much greater benefits compared with TCP.

In addition, we also measure the loads on each connection in the scenario where there are 5 auxiliary TCP/UDP connections. In the linear(campus) network, each switch is connected to 60(30) hosts. The simulation results, as shown in Table 1, show that the

Table 1: The load (MB) of connections (connection #0 is the main connection, the others are auxiliary).

	0	1	2	3	4	5
TCP (linear)	0.37	0.86	0.87	0.86	0.85	0.88
UDP (linear)	0.37	0.86	0.87	0.86	0.85	0.88
TCP (campus)	2.38	4.52	4.51	4.49	4.48	4.56
UDP (campus)	2.37	4.48	4.47	4.47	4.46	4.53

loads of the auxiliary connections are well balanced. This is because Packet-In and Packet-Out messages contribute to most of the traffic between the controller and switches in our simulations.

These experiments only show the power of auxiliary connections in alleviating the congestion pressure of the main connection. More benefits can be achieved if the switch parallelism can be exploited using the auxiliary connections. For example, parallel packet pipeline processing can improve the throughput of the switch and decrease the corresponding delay.

5 CONCLUSIONS AND FUTURE WORK

This work has introduced an extension of OFSWITCH13 to support multiple TCP/UDP auxiliary connections in SDN simulations based on ns-3. With this extension, experiments can be conducted to exploit the potential of auxiliary connections in improving SDN performance. In addition, we presented the simulation results of linear topology and campus topology with auxiliary connections, which shows the benefits and limits of the auxiliary connections. Based on this work, new scheduling algorithms for spreading various packets across the various connections can be verified. Furthermore, a comprehensive examination of the impact of connection priority, connection type, controller, applications, and other factors in terms of more metrics (e.g., packet delay) is required. What's more important, how to exploit switch parallelism with auxiliary connections to build more scalable SDN switches will be a rich area of future research.

REFERENCES

[1] A. Basta, A. Blenk, H. Belhaj Hassine, and W. Kellerer. 2015. Towards a Dynamic SDN Virtualization Layer: Control Path Migration Protocol. In *2015 11th International Conference on Network and Service Management (CNSM)*. Barcelona, Spain, 354–359. DOI: http://dx.doi.org/10.1109/CNSM.2015.7367382

[2] M. Chan, C. Chen, J. Huang, T. Kuo, L. Yen, and C. Tseng. 2014. OpenNet: A Simulator for Software-defined Wireless Local Area Network. In *Wireless Communications and Networking Conference (WCNC), 2014 IEEE*. Istanbul, Turkey, 3332–3336. DOI: http://dx.doi.org/10.1109/WCNC.2014.6953088

[3] L. Jerez Chaves, I. Calciolari Garcia, and E. R. Mauro Madeira. 2016. OFSwitch13: Enhancing ns-3 with OpenFlow 1.3 Support. In *Proceedings of the Workshop on ns-3 (WNS3 '16)*. ACM, Seattle, WA, USA, 33–40. DOI: http://dx.doi.org/10.1145/2915371.2915381

[4] A. Kumbhare et al. 2017. *OpenDaylight Openflow Plugin.* https://github.com/opendaylight/openflowplugin.

[5] A. R. Curtis et al. 2011. DevoFlow: Scaling Flow Management for High-Performance Networks. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 254–265.

[6] B. Pfaff et al. 2012. *OpenFlow Switch Specification, Version 1.3.0 (Wire Protocol 0x04).* https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf.

[7] E. Leao Fernandes et al. 2014. *OpenFlow 1.3 Software Switch.* http://cpqd.github.com/ofsoftswitch13.

[8] K. Kaplita et al. 2015. *LINC - OpenFlow Software Switch.* https://github.com/FlowForwarding/LINC-Switch.

[9] R. Lzard et al. 2017. *Floodlight OpenFlow Controller (OSS).* https://github.com/floodlight/floodlight.

[10] M. Gupta, J. Sommers, and P. Barford. 2013. Fast, Accurate Simulation for SDN Prototyping. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*. ACM, Hong Kong, China, 31–36. DOI: http://dx.doi.org/10.1145/2491185.2491202

[11] Pica8 Inc. 2015. *PicOS Open VSwitch Configuration Guide.* http://www.pica8.com/document/v2.6/html/ovs-configuration-guide/.

[12] J. Ivey, H. Yang, C. Zhang, and G. Riley. 2016. Comparing a Scalable SDN Simulation Framework Built on ns-3 and DCE with Existing SDN Simulators and Emulators. In *Proceedings of the 2016 Annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation (SIGSIM-PADS '16)*. ACM, Banff, Alberta, Canada, 153–164. DOI: http://dx.doi.org/10.1145/2901378.2901391

[13] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia. 2011. Modeling and Performance Evaluation of an OpenFlow Architecture. In *Proceedings of the 23rd International Teletraffic Congress (ITC '11)*. International Teletraffic Congress, San Francisco, CA, USA, 1–7. http://dl.acm.org/citation.cfm?id=2043468.2043470

[14] Z. Li, Q. Li, L. Zhao, and H. Xiong. 2014. Openflow Channel Deployment Algorithm for Software-Defined AFDX. In *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*. Seattle, WA, USA, 4A6–1–4A6–10. DOI: http://dx.doi.org/10.1109/DASC.2014.6979466

[15] R. Pan, B. Prabhakar, K. Psounis, and D. Wischik. 2005. SHRiNK: a Method for Enabling Scaleable Performance Prediction and Efficient Network Simulation. *IEEE/ACM Transactions on Networking* 13, 5 (Oct 2005), 975–988. DOI: http://dx.doi.org/10.1109/TNET.2005.857080

[16] B. Yeong Yoon. 2015. Method and Apparatus for Network Failure Restoration. (Jan. 20 2015). US Patent App. 14/600,892.