

quicSDN: Transitioning from TCP to QUIC for Southbound Communication in SDNs

Puneet Kumar and Behnam Dezfouli

Internet of Things Research Lab, Department of Computer Science and Engineering, Santa Clara University, USA
 {pkumar, bdezfouli}@scu.edu

Abstract—In Software-Defined Networks (SDNs), the control plane and data plane communicate for various purposes, such as applying configurations and collecting statistical data. While various methods have been proposed to reduce the overhead and enhance the scalability of SDNs, the impact of the transport layer protocol used for southbound communication has not been investigated. Existing SDNs rely on TCP (and TLS) to enforce reliability and security. In this paper, we show that the use of TCP imposes a considerable overhead on southbound communication, identify the causes of this overhead, and demonstrate how replacing TCP with QUIC can enhance the performance of this communication. We introduce the quicSDN architecture, enabling southbound communication in SDNs via the QUIC protocol. We present a reference architecture based on the standard, most widely used protocols by the SDN community and show how the controller and switch are revamped to facilitate this transition. We compare, both analytically and empirically, the performance of quicSDN versus the traditional SDN architecture and confirm the superior performance of quicSDN.

Index Terms—Software-defined Networks, overhead, latency, UDP, RYU, OVS, agents

I. INTRODUCTION

Software-Defined Networks (SDNs) simplify new application development and facilitate network monitoring and management. Nowadays, SDN architectures are being used in various types of deployments, such as data center networks, Wide Area Networks (WANs) [1], Network Function Virtualization (NFV) [2], 5G [3], and edge and fog computing [4], [5].

The two primary components of a SDN architecture are controller(s) and switch(es). A logically centralized controller implements the control plane, and switches implement the data plane. A controller, such as Ryu [6] and OpenDayLight (ODL) [7], provides functionalities such as network topology discovery, network operation analysis, and flow rule computation and installation. The controller provides northbound Application Programming Interfaces (APIs) to facilitate the development of various applications such as intrusion detection and load balancing. Communication between the controller and the switches is achieved via *southbound interfaces* ranging from traditional protocols such as Simple Network Management Protocol (SNMP) [8] to more advanced ones such as OpenFlow [9], Open vSwitch Database (OVSDB) [10], and NETCONF (Network Configuration Protocol) [11].

The introduction of SDN allows for fine-grained and centralized control over the operation of the data plane. For example, OpenFlow and its vendor-specific flavors such as Arista's DirectFlow [12], Cisco-OpenFlow-Plugin [13], HP OpenFlow

[14]) are being used to configure flow rules and flow tables in switches via exchanging variety of messages. The three main message types exchanged between a controller and a switch via the OpenFlow protocol are `Packet_in`, `Flow_mod`, and `Multipart_request/reply`. When a data packet arrives at a switch and does not match the existing installed flow rules, the packet is forwarded to the controller via a `Packet_in` message. In large networks, such as data centers, an enormous amount of `Packet_in` messages are generated from table misses [15]–[17]. This is primarily caused by the limited memory of switches and the arrival of new flows [18]. As the table miss rate increases, the overhead of transport layer protocol used for communication between the controller and switches elevates [19], [20]. `Flow_mod` packets, which are used to install or modify flow rules, can be sent reactively in response to a `Packet_in` message or proactively in anticipation of expected traffic. For example, the controller may proactively install flow rules based on the collected statistics from switches to address requirements such as load balancing. To maintain an up-to-date view of network status, a controller needs to poll switches at regular intervals. The controller sends `Multipart_request` messages to each switch for each feature that it needs to collect statistics on, and each switch responds with a corresponding `Multipart_reply` message. The polling frequency depends on the types of applications running on the controller. The reply messages' size is variable and depends on the switch configuration [21]. For instance, switches with many queues and large flow tables transmit several messages for each poll event. Another protocol, OVSDB, is used to configure Quality of Service (QoS) functionalities via Remote Procedure Calls (RPC) "transact" interactions. Configuring a queue on a switch involves multiple OVSDB messages: (i) the queue is added to the switch, (ii) the queue is added to a specific packet scheduler, and (iii) the switch responds with an RPC "update" to confirm the new configurations. More messages are required for more complicated switch operations, and the overhead is even more significant [21]–[26].

In addition to the basic functionalities offered by OpenFlow and OVSDB, over the past few years and towards the development of next-generation networks, the capability and flexibility of network switches, smartNICs, and middleboxes have considerably increased to run various configuration and management protocols. To enhance the scalability and development of new applications and services, it is essential to provide end-to-end traffic engineering (resource allocation and security), fault detection, recovery, and isolation. Programmability and

automation are necessary to provide such features in a scalable and dynamic fashion, and observability is essential to react to network dynamics. To adapt to the needs of large-scale data-centers, campus networks, and carrier networks, modern switches and devices running Network Operating Systems (NOSs) offer more flexibility such as programmability of data plane via P4 [27], APIs for accessing and configuring switches' data plane state (e.g., OpenConfig [28], NETCONF [11]), event management, Linux shell access, and the capability to run containers and Virtual Machines (VMs). For example, Arista's Extensible Operating System (EOS) [29] provides a Software Development Kit (SDK) for development of programs, referred to as *agents*, that can access the status and configure the operation of switches. These agents, which can be dynamically added to or removed from the system, can implement custom protocols or rely on open protocols. For example, gRPC Network Management Interface (gNMI) is used for configuration and telemetry, gRPC Network Operations Interface (gNOI) is used for exchanging operational commands, and gRPC Interface to a Network Element RIB (gRIBI) is used for exchanging commands pertaining to routing table control. For instance, one can develop and run an agent, which continuously monitors and generates reports (including low-level counters and system temperature) that allow the controller to implement machine learning algorithms for device failure detection.

The wide range of programmability features available in today's switches reveals the increasing demand for southbound communication in SDNs. However, this communication imposes significant transport layer protocol overhead that severely affects the utilization of bandwidth resources and network scalability [30], [31]. In this paper, we look at the communication between the control plane and data plane from a different perspective—the *transport-layer protocols*. Currently, communication reliability, packet reordering, and security in SDNs are achieved by using Transmission Control Protocol (TCP) (combined with Transport Layer Security (TLS)). Throughout this paper, we use the term *tcpSDN* to refer to SDNs that utilize TCP as the transport layer protocol used for southbound communication. *tcpSDN* architectures introduce various shortcomings in terms of communication overhead, lack of connection multiplexing, and high overhead of connection establishment. For example, when multiple agents run on a switch communicating with a controller, each agent needs to open its own connection due to the lack of supporting multiplexing connections by TCP, thereby increasing communication overhead. Also, TCP does not deal with the Head-of-Line (HOL) blocking problem, which causes increased message delivery delay.

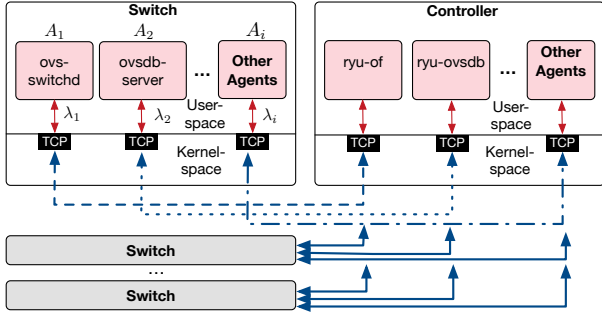
In this paper, we introduce *quicSDN*, a novel framework to address the drawbacks and challenges of using TCP as the transport layer protocol for southbound communication in SDNs. Specifically, instead of using TCP, *quicSDN* uses a new transport layer protocol called Quick UDP Internet Connection (QUIC) [32]. Although QUIC was primarily designed for web traffic (HTTP3 [33]), its enhancements over TCP are applicable across various domains. These enhancements include the ability to multiplex different streams, reduction in connection

establishment latency, mitigation of the HOL blocking problem, and adding the capability to differentiate between the ACK sent for original and retransmitted packets (a.k.a., TCP ambiguity problem). Towards proposing a novel architecture for SDNs, this paper presents the following contributions:

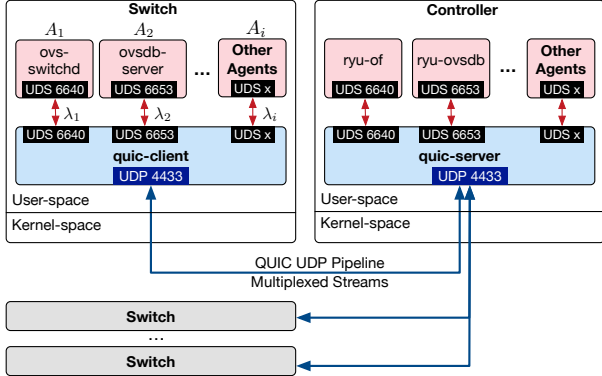
- We first identify and analyze the benefits of using QUIC instead of TCP for southbound communication. In addition to highlighting the benefits of QUIC considering the specific properties of SDNs, we present an analytical modeling of overhead when multiple agents on a switch communicate with one or multiple agents on a controller. In particular, this analysis shows that the overhead of QUIC drops compared to TCP as we increase the number of agents on the switch or reduce inter-message generation intervals. Also, we justify the mitigation of HOL problem and discuss the benefits of faster connection establishment by QUIC.
- Towards providing a framework for transitioning from *tcpSDN* to *quicSDN*, we discuss the design options and present a system architecture based on RYU controller and switches running Open vSwitch (OVS) and OVSDB. The proposed framework details aspects such as understanding and removing the intertwined dependency of RYU, OVS, and OVSDB on TCP and replacing them with QUIC. Additionally, the proposed framework details the Inter-Process Communication (IPC) methods to allow RYU, OVS, and OVSDB to communicate through QUIC. It is worth mentioning that the proposed framework can be used to integrate additional southbound protocols (e.g., NETCONF) and controllers (e.g., OpenDaylight). The implementation of the proposed framework is publicly and freely available.¹ This implementation includes newly developed and modified entities of an SDN architecture, which are RYU controller, OVS switch, QUIC protocol, and their dependent third party libraries such as Eventlet [34] and Libevent [35]. We believe that this framework enables the research community to repeat our results and extend the functionalities of *quicSDN*. Furthermore, this will lay a foundation for the evolution of QUIC protocol in the context of SDN architectures.
- We built a testbed to empirically evaluate *quicSDN* versus *tcpSDN*. We evaluate control traffic overhead versus varying message rate, loss rate, and number of streams. The results confirm the lower communication overhead of *quicSDN* versus *tcpSDN* in all the scenarios. Also, we measure message delivery delay between controller and switches and confirm the superior performance of *quicSDN*.

The rest of this paper is organized as follows: Section II presents an analytical study of the communication overhead of QUIC and TCP and highlights the main differences between the two protocols. The architecture of *quicSDN* is presented in Section III. Section IV discusses the implementation, algorithms, and pertinent details of *quicSDN*. Empirical evaluations are given in Section V. Section VI overviews the related work. We conclude the paper in Section VII.

¹<https://github.com/SIOTLAB/quicSDN>



(a) tcpSDN Architecture.



(b) quicSDN Architecture.

Fig. 1: tcpSDN (a) and quicSDN (b) architectures. Compared to the tcpSDN architecture where an individual connection is required for each pair of agents (processes) running on a switch and the controller, quicSDN establishes one connection between each switch and the controller.

II. MOTIVATION

In this section, we highlight and analyze the main differences between TCP and QUIC and justify the benefits of QUIC for southbound communication in SDNs. For this analysis, we also develop mathematical models to showcase the benefits of quicSDN over tcpSDN.

A. Communication Overhead

TCP is the widely-used transport protocol to ensure packet ordering and reliability in end-to-end message delivery. Despite its prevalence, TCP has several major shortcomings. In this section, we focus on packet transmission overhead.

Figure 1a shows the tcpSDN architecture, which represents existing SDNs that use TCP as the transport protocol for southbound communication. Multiple *agents*, a.k.a., processes or applications (such as OpenFlow and OVSDB), running on a switch need to communicate with the controller. We denote the list of agents as $\mathbf{A} = \{A_1, A_2, \dots\}$ and the number of agents as $|\mathbf{A}|$. Using TCP, each of these agents must establish its own connection, which means the overheads pertaining to connection establishment and data exchange scale with the number of connections. Consider the following scenario to model the overhead of data exchange between two devices. Each agent A_i generates message set $\mathbf{M}_{A_i} = \{m_1^i, m_2^i, \dots\}$, where each element of the set represents the message size in bytes. Also, assume the intervals between message generations

TABLE I: Header size values used for the evaluations of this section.

Headers	Size (Byte)
Ethernet and Physical Headers (H_{EP})	28
IP Header (H_{IP})	20
TCP Header (H_{TCP})	20
UDP Header (H_{UDP})	8
QUIC Short Header (H_{QSH})	2
QUIC Frame Header (H_{QFH})	1

are short enough to place consecutive messages in a packet as long as there is available room. Additionally, we assume the congestion window and receive window do not cause reduction in throughput, and there is no packet loss. Using TCP, *each agent must be associated with its own socket*, as Figure 1a shows. Therefore, the minimum overhead of sending data from the sender to the receiver is:

$$\sum_{\forall A_i \in \mathbf{A}} ((H_{EP} + H_{IP} + H_{TCP}) \times \left[\frac{\sum_{\forall m_j^i \in \mathbf{M}_{A_i}} m_j^i}{L_{MTU} - (H_{IP} + H_{TCP})} \right]) \quad (1)$$

where m_j^i refers to the size of message j generated by agent A_i , L_{MTU} is Maximum Transmission Unit (MTU) size, H_{EP} is physical layer and Ethernet header size (including inter-packet gap), H_{IP} is IP header size, and H_{TCP} is TCP header size (without any options field). The values for these parameters can be found in Table I.

In this paper, we propose and develop the *quicSDN* architecture, demonstrated in Figure 1b.² This architecture relies on the fact that multiple agents on each switch need to communicate with a controller. Therefore, instead of establishing individual connections between *each agent and the controller*, quicSDN establishes one connection between *each switch and the controller*.³ The underlying QUIC protocol *multiplexes* multiple connections between two endpoints and converts them into *streams* inside a User Datagram Protocol (UDP) pipeline. A stream is formatted as a *frame* inside a packet and represents a lightweight abstraction of server-client connection and is uniquely identified by a connection ID (CID). Except during the connection establishment phase, each QUIC packet includes a QUIC Short Header (QSH) (denoted as H_{QSH}), and there is a QUIC Frame Header (QFH) (denoted as H_{QFH}) for each frame included in the packet. For the scenario given above, using QUIC results in the following minimum overhead:

$$(H_{EP} + H_{IP} + H_{UDP} + H_{QSH} + \alpha \times H_{QFH}) \times \left[\frac{\sum_{\forall A_i \in \mathbf{A}} \sum_{\forall m_j^i \in \mathbf{M}_{A_i}} m_j^i}{L_{MTU} - (H_{IP} + H_{UDP} + H_{QSH} + \alpha \times H_{QFH})} \right] \quad (2)$$

where α is the number of frames per packet, which depends on the size of messages and their generation pattern.

To simplify the analysis for determining the value of α , we consider two cases, depending on the average message

²We will explain the implementation details in Sections III and IV.

³The same concept applies to other data-plane components such as middle-boxes and smart NICs that need to communicate with a controller.

size. The maximum available space per packet for including messages is $L_{max} = L_{MTU} - (H_{EP} + H_{IP} + H_{UDP} + H_{QSH} + H_{QFH})$. If the average message size is larger than L_{max} , each packet includes either one or two frames. For example, if the average message size is 1800 bytes, the first packet sent includes one frame (part of this message), and the second packet consists of two frames, which are the residual of the first message and the first part of the second message. If the average message size is less than L_{max} , then α is computed as follows:

$$\operatorname{argmax}_{\alpha} \left(\frac{L_{max} + H_{QFH}}{\alpha \times (m_{avg} + H_{QFH})} > 1 \right), \text{ and } \alpha \in \mathbb{N}. \quad (3)$$

Equations 1 and 2 can be used to compute overhead, neglecting inter-message generation intervals. To represent a realistic scenario, we assume each agent (A_i) generates traffic at rate λ_{A_i} messages per second. This system represents $|\mathbf{A}|$ independent exponential random variables, where $|\mathbf{A}|$ is the number of agents. To enhance the efficiency of message transmissions, transport protocols use a buffering period during which the transport layer is awaiting additional data from the application layer. For example, Linux includes an implementation of Nagle's algorithm [36], which waits for more data from the application layer when there is pending ACK and the amount of data for transmission is less than Maximum Segment Size (MSS). We refer to the buffering delay in the transport layer as T_b . Therefore, to compute communication overhead, we need to model the effect of message generation burstiness arriving in the transport protocol. Since the interval between message arrivals follows the exponential distribution, the expected number of messages generated by an agent A_i during the buffering time is $\lambda_{A_i} \times T_b$. Assuming that all the messages are equal size ($\bar{m} = m_1^i = m_2^i = \dots, \forall A_i \in \mathbf{A}$) and the message generation rate of all the agents is the same ($\lambda = \lambda_{A_i}, \forall A_i \in \mathbf{A}$), we can use message burstiness probabilities to compute the number of messages, and therefore the number of bytes generated per burst. Let $\bar{B}_{A_i} = \{b_1, b_2, \dots\}$ represent the list of message bursts generated by agent A_i . Each element b_j is the number of bytes in a burst. The effect of burstiness on communication overhead of TCP is presented as follows:

$$\sum_{\forall A_i \in \mathbf{A}} \sum_{\forall b_j \in \bar{B}_{A_i}} ((H_{EP} + H_{IP} + H_{TCP}) \times \left\lceil \frac{b_j}{L_{MTU} - (H_{IP} + H_{TCP})} \right\rceil). \quad (4)$$

As the message generation rate of each agent increases, the header overhead of TCP drops because data bytes belonging to different messages can be included in each packet, thereby sending larger packets. However, QUIC performs a better job in aggregating multiple messages and sending larger packets instead of multiple smaller packets, as we show in the following.

The quicSDN architecture allows multiple agents to use a single connection to communicate with one or more processes on the controller. Therefore, the overall rate of incoming messages into the QUIC protocol (quic-client or quic-server

in Figure 1b) is higher than TCP. Specifically, the inter-message time can be represented as the sum of $|\mathbf{A}|$ independent exponential variables. We represent this accumulated rate as $\hat{\lambda} = \sum_{\forall A_i \in \mathbf{A}} \lambda_{A_i}$. With the accumulated incoming message rate $\hat{\lambda}$, we generate the list of message bursts as $\hat{\mathbf{B}} = \{\hat{b}_1, \hat{b}_2, \dots\}$, where each element \hat{b}_j is the number of bytes in a burst. The overhead of QUIC is computed as follows:

$$\sum_{\forall \hat{b}_j \in \hat{\mathbf{B}}} ((H_{EP} + H_{IP} + H_{UDP} + H_{QSH} + \alpha \times H_{QFH}) \times \left\lceil \frac{\hat{b}_j}{L_{MTU} - (H_{IP} + H_{UDP} + H_{QSH} + \alpha \times H_{QFH})} \right\rceil). \quad (5)$$

We use the models presented in this section to compare the transmission overhead of TCP and QUIC. Transmission overhead represents the overhead associated with the layers of the protocol stack (i.e., headers and Ethernet's inter-packet gap) when sending a certain number of messages. We also compute the overhead of ACK packets sent from a receiver to a sender as follows. Neglecting the effect of packet loss and variable RTT, for a TCP connection, the number of ACK packets sent depends on the number of data packets received from the sender: the receiver either sends an ACK immediately or waits up to 500 ms to receive a second packet and then send an ACK. Assume the number of data packets sent is denoted as d , the mean number of ACK packets sent is $(d + d/2)/2$. Each ACK packet includes a TCP header, in addition to the headers of underlying layers. A similar method is used to compute the ACK overhead of QUIC.

Figure 2 presents the results when we vary message rate (λ) and the number of agents ($|\mathbf{A}|$). The average message size (\bar{m}) is 500 bytes. The total number of messages generated per agent is 5000. These results show that as the buffering delay (T_b) increases, the difference between the overhead (and the number of packets) of TCP and QUIC reduces until they reach their minimum values. The lower acknowledgment overhead of QUIC is a direct effect of a lesser number of packets sent by this protocol. Since QUIC can multiplex multiple agents' messages into one connection, its communication overhead declines and stabilizes faster than TCP. Especially, as the number of agents increases, the decline rate of QUIC's overhead increases as well, and this can be observed by comparing the first column with the second column and the third column with the fourth column.

It must be noted that the multiplexing feature of quicSDN comes at a cost, associated with attaching QFH to each frame in a packet. Specifically, the overhead of QUIC is higher than TCP when $H_{UDP} + H_{QSH} + \alpha \times H_{QFH} > H_{TCP}$. For example, considering the parameters given in Table I, the overhead of quicSDN is higher than tcpSDN when more than ten streams are included in a packet. Nevertheless, quicSDN results in a lower number of packet transmissions by establishing a single connection to carry the data of all the agents. We will empirically evaluate the overhead of quicSDN and tcpSDN in Section V-A.

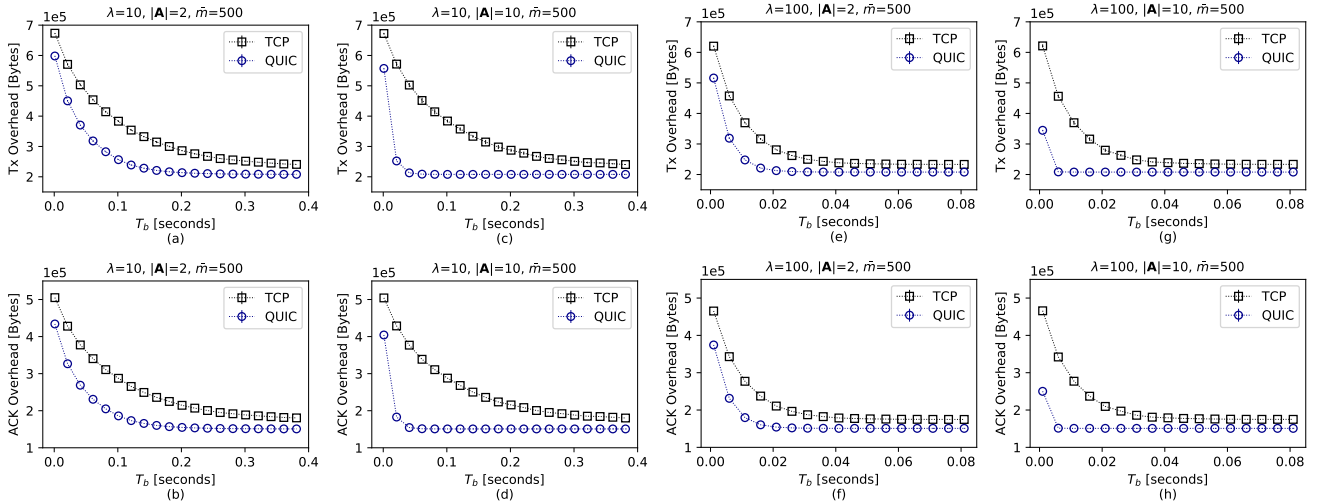


Fig. 2: Each column shows the transmission overhead and acknowledgement overhead of TCP and QUIC in various scenarios. The first row shows transmission overhead and the second row shows acknowledgement overhead. λ is rate of message generation by each agent. $|A|$ is the number of agents. \bar{m} is average message size. These results confirm the considerably lower packet exchange overhead of QUIC versus TCP.

B. Head of Line Blocking Problem

The use of multiplexing allows QUIC to mitigate the HOL blocking problem. In TCP, when packets arrive out of order at the receiver, since the protocol is unaware of the boundary between messages, it cannot deliver completely received messages to the application layer. Whereas, QUIC reduces message delivery delay by assigning messages to streams. For example, consider the scenario given in Figure 3, where a controller (sender) sends two messages, Message k and Message $k+1$ to a switch (receiver). The transmission of Message k is performed by sending n packets. Packet i includes part $n-2$ of Message k and is successfully received by the receiver at time t_1 . Packet $i+1$, which includes part $n-1$ of Message k , is lost. Packet $i+2$ includes the last part of Message k and all the bytes of Message $k+1$. At time t_2 , although Message $k+1$ has been fully received, TCP does not deliver it to the application layer. In contrast, QUIC can use frame headers to identify message boundaries, and therefore, Message $k+1$ is delivered to the application layer as soon as Packet $i+2$ is received. The retransmission of Packet $i+1$ is triggered by the Retransmission Timeout (RTO) of sender; alternatively, assuming that more packets are sent after packet $i+2$, the reception of three duplicate ACKs triggers the TCP fast retransmission method to retransmit Packet $i+1$. In either case, the delivery delay of Message $k+1$ to the application is at least one Round Trip Time (RTT) delayed when using TCP, compared to QUIC. The shorter message delivery delay of QUIC is beneficial in SDNs. For example, if Message $k+1$ is a flow rule, quicSDN provides faster reaction to new flow arrival into a switch. We will empirically evaluate the effect of HOL in Section V-B.

C. Connection Establishment and Migration

Both TCP and QUIC need to establish a connection before data exchange. TCP with TLS1.2 [37] and TLS1.3 [38] require three and two RTTs, respectively, for connection establish-

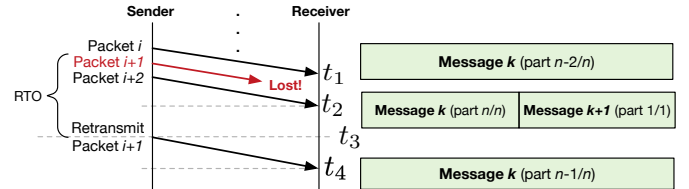


Fig. 3: The effect of HOL blocking on message delivery delay to application layer. In this scenario, the delivery of Message $k+1$ is at least $RTT + RTT/2$ slower when using TCP, compared to QUIC.

ment. By leveraging a multi-stage key exchange, QUIC combines the transport and security layer connection establishment procedures to minimize connection establishment overhead to one RTT. In the first stage, the client sends a 'hello' message (CHLO) to retrieve the server's configuration. Since the client is unknown to the server, the server responds with a REJ packet. The REJ packet contains the server's configuration, long term Diffie-Hellman value, key agreement, `cid`, and initial data. The client then authenticates the server by verifying the certificate chain and signature. After authentication, the client sends a complete CHLO packet to the server and finishes the first handshake. At this stage, the client has the initial keys and is ready to exchange application data with the server. Upon a successful first handshake, the server sends a complete hello (SHLO) to the client and concludes the final handshake. To support connection migration, QUIC uses a unique `cid` to identify each connection. This allows for connection rebinding even if the connection parameters such as IP address or port number are changed. Therefore, if a switch is assigned to a controller that it has communicated with in the past, quicSDN can establish the connections in zero RTT, while TCP with TLS1.3 requires one RTT.

The shorter connection establishment time of QUIC allows quicSDN to provide the following benefits. First, if a switch detects connection drop with a controller, the delay of connection reestablishment with the same or another controller

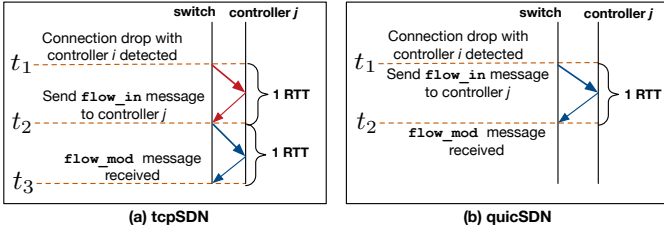


Fig. 4: Connection reestablishment delay between switches and controllers in (a) tcpSDN and (b) quicSDN. quicSDN facilitates dynamic assignment of switches to controllers with a shorter delay and lower overhead.

is lower compared to tcpSDN. A sample scenario is given in Figure 4. Assume at time t_1 the switch detects connection drop with controller i . This is detected, for example, when the switch sends a `flow_in` to the controller, but no response is received within a timeout period. At this point, if using tcpSDN, the switch needs to first establish a connection with controller j (during t_1 to t_2). Then, at time t_2 , the `flow_in` message is sent to controller j . This process requires $2 \times \text{RTT}$. In contrast, quicSDN eliminates the connection reestablishment delay and immediately sends the `flow_in` message to controller j at time t_1 . Thereby, the delay of communication with controller j is reduced to RTT. The lower delay of connection reestablishment in quicSDN also reduces the overhead of (proactive) switch reassignment to controllers (e.g., when the load of controllers are periodically balanced). Therefore, the quicSDN framework proposed in this paper facilitates the development of enhanced load balancing and controller assignment solutions, building on top of the methods proposed in existing works [39]–[42].

D. Congestion and Flow Control

QUIC’s congestion control mechanism provides a richer set of features compared to TCP [43]. For instance, consider the TCP ambiguity problem, where TCP cannot determine if the ACK was for the original or retransmitted packet. QUIC solves this problem by assigning a unique *Packet Number* to each packet, irrespective of being an original or a retransmission. QUIC also reduces congestion control by using a Negative Acknowledgement (NACK) scheme, where, instead of acknowledging every packet, the receiver notifies the sender about lost packets [44].

In TCP, a sender can be blocked from sending data when the entire receiver buffer is consumed. QUIC addresses this problem via two methods: First, with connection-level flow control, in which an upper-limit is imposed on the entire connection for a receiver’s aggregated buffer. Second, flow-level flow control imposes an upper-limit on the flow-level buffer size on the receiver. To reduce or increase flow-level buffer size, QUIC uses a window update frame for advertising per-stream absolute byte offsets for received, delivered, and sent packets. These per-stream absolute byte offsets dictates the amount of bytes a receiver is willing to accept on a particular stream.

III. QUICSDN ARCHITECTURE

This section presents a high-level overview of the interactions between the components of quicSDN: QUIC, OVS, and RYU. In typical scenario, QUIC is used by an application by incorporating QUIC’s code into the application’s code and compiled as one agent. This prohibits the use of single QUIC instance by multiple applications. The memory allocation performed for queues and buffers of QUIC is restricted to the application it was compiled with. The quicSDN architecture is different than this method because, on a device (switch or controller), multiple agents (processes) interact with a QUIC instance. As Figure 1b shows, `ovsdb-server`, `ovs-switchd`, and `quic-client` run on the switch, and `ryu-ovsdb`, `ryu-of`, and `quic-server` run on the controller. `ovsdb-server` and `ovs-switchd` are agents responsible for processing OVSDB and OpenFlow packets on the switch side. `ryu-ovsdb` and `ryu-of` are agents running on the controller to process the OVSDB and OpenFlow packets respectively. `quic-client` and `quic-server` are agents running on the switch and controller, respectively, to establish communication between the switch and the controller.

A. Inter-process Communication (IPC)

Since QUIC is an application-layer protocol, it cannot be used as an operating system’s inbuilt transport protocol (like TCP or UDP). Therefore, an IPC is required to facilitate communication between QUIC and application processes. This section describes the pros and cons of various IPC methods for quicSDN.

1) *Shared Memory*: To allow multiple applications (agents) to communicate through shared data structures, either they must be compiled as a single application, or they can use shared memory via a memory map. One of the main drawbacks of these methods is the lack of extensibility and abstraction. Specifically, accessing the source code of all the modules is necessary to implement these methods. For example, suppose there is a plan to extend a switch’s features by adding a component; in that case, its code must be fully available to be integrated with the existing ones. Also, even when the new component’s source code is available, the developer still needs to understand the execution paths of the code thoroughly. For instance, code modification and the introduction of new threads are usually required to allow concurrent execution of components. Furthermore, the larger code size and the lack of clear interfaces between modules result in more complicated code debugging and enhancements when employing these methods. Additionally, accessing shared data structures also causes race conditions. It is essential to acquire mutually exclusive locks to avoid race conditions among message producers and consumers. These locks can cause performance bottleneck by introducing differences in the rate of packets processed by switches or controller. This observation has been made in multiple studies [45], [46].

2) *Message Passing*: Compared to shared memory, message passing methods are easier to implement and more extensible. The two primary methods of message passing are message queues and Unix Domain Socket (UDS). To simplify

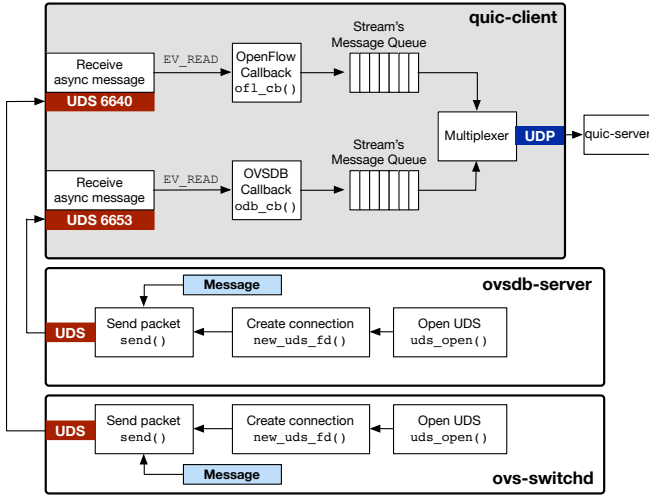


Fig. 5: The architecture of a switch in quicSDN. The figure highlights the modifications to OVS and how packets are processed by the quic-client component.

system extensibility, we use the latter method because of its ease of debugging and support in all the major programming languages. There are two primary types of UDSs at the transport layer level: stream sockets and datagram sockets. With stream UDS, received data is in form of stream bytes. The arrival of stream bytes can be out-of-order and it is required to be put in order by the application based on message boundaries. Finding message boundaries in stream sockets introduces processing overhead because they are byte oriented and the receiver needs to parse and rearrange the received bytes, thereby introducing additional overhead. On the other hand, datagram sockets are faster and allow an entire message to be passed.

B. Switch

Within the quicSDN architecture, multiple processes on a switch can communicate with the controller via the quic-client module. There are two entities that communicate with quic-client: *ovs-switchd* and *ovsdb-server*, handling OpenFlow and OVSDB, respectively. Figure 5 presents the architecture of a quicSDN switch.

On *tcpSDN* switches, *ovs-switchd* and *ovsdb-server* are connected to the controller using the following Command Line Interface (CLI) commands:

```
$ ovs-vsctl set-controller <bridge name>
  tcp:<controller-IP>:<port>
$ ovs-vsctl set-manager tcp:<controller-IP>:<port>
```

The quicSDN architecture provides new CLI commands to allow *ovsdb-server*'s and *ovs-switchd*'s UDSs to communicate with quic-client.

```
$ ovs-vsctl set-controller <bridge name>
  udp:<controller-IP>:<port>
$ ovs-vsctl set-manager udp:<controller-IP>:<port>
```

To enable the new interface, we developed the `udp_vconn_class` class and its associated function pointers to search for the "udp" keyword in the CLIs and open the UDP connections to quic-client. The opened connections are mapped to stream pointer File Descriptors

(FDs), which are registered in function `new_uds_fd()`. The aforementioned process is for both *ovsdb-server* and *ovs-switchd*.

quic-client spawns two UDP servers listening on ports 6653 and 6640. The messages received on these ports are processed and multiplexed in *quic-client* and then transmitted to the *quic-server*. To avoid any thread blockage while waiting for packet arrivals, we use async I/O operations by leveraging the *libevent* library, a concurrent, highly scalable network library. Libevent library attaches a callback function to a FD associated to an application. This callback function is invoked and notifies the application if an event occurs on the FD, such as receiving or sending data. The two newly-introduced FDs for sockets, along with their callbacks, on ports 6653 and 6640 are mapped to stream pointers in *quic-client* to communicate with *ovsdb-server* and *ovs-switchd*. The QUIC RFC [32] mandates the use of even and odd stream IDs for client-initiated and server-initiated connections, respectively. In order to distinguish packets received on ports 6653 and 6640 on *quic-client*, different stream IDs are selected for OpenFlow and OVSDB. Since all stream IDs of client-initiated connections in *quic-client* are even, we assign even stream IDs divisible by 3 for messages sent to *ryu-of* and the rest are used for messages sent to *ryu-ovsdb*. This assignment happens in the round robin fashion for the available opened streams. Packets generated on the stream IDs are multiplexed into the same UDP pipeline for transmission to *quic-server*.

C. Controller

Figure 6 shows the controller architecture. The two main entities of quicSDN's controller are *quic-server* and RYU. The RYU entity includes two agents, *ryu-of* and *ryu-ovsdb*, which communicate with *quic-server* over a datagram connection on ports 6653 and 6640.

The RYU controller's asynchronous I/O infrastructure is based on the *eventlet* library, which is a highly scalable and non-blocking I/O library. The *eventlet* library socket implementation is different than the standard `socket.socket` class in Python. The *eventlet* library implements sockets as *GreenSockets* [47] and sets them into a non-blocking state to support asynchronous I/O operations. RYU spawns a server based on *GreenSocket* and registers an event loop to receive data on the *GreenSocket*. In order to make it UDP compatible, the event loop is modified by dismantling all the TCP related code and modifying the callbacks.

After receiving packets from *quic-client*, *quic-server* performs demultiplexing by disassembling streams based on their IDs. If the stream ID is divisible by 3, then the packet is delivered to *ryu-of*, otherwise it is delivered to *ryu-ovsdb*.

IV. IMPLEMENTATION

This section presents the newly developed and modified entities in quicSDN. We use color-coding schemes to highlight the newly-developed and modified entities. Blue highlights indicate newly-developed entities, and the red highlights indicate modified entities. We use three programming languages in our implementations: C++ for QUIC, Python for Ryu, and C for

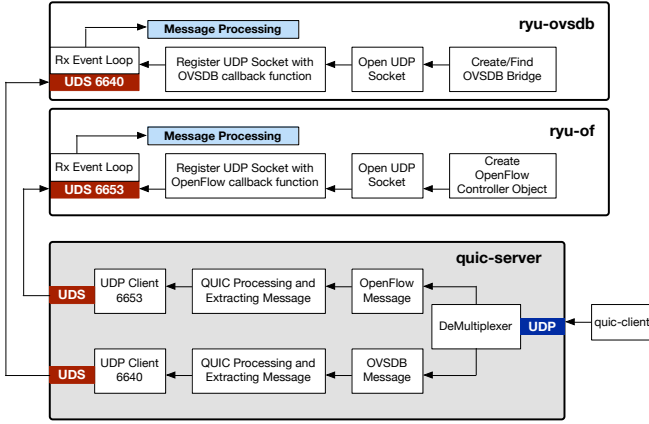


Fig. 6: The architecture of controller in quicSDN. This figure highlights the modifications to RYU and shows how packets are processed by the quic-server component.

OVS. We use these languages to meet the varying performance and programmability requirements of different SDN software components. For example, we use Python to program the controller because it simplifies application development and extensibility. On the switch, we use C language because our main focus is to enhance performance.

A. OVS

This section describes the modifications made to OVS to make it compatible with quicSDN. With tcpSDN, the OpenFlow and OVSDB protocols used by OVS are implemented on TCP, sitting in the Linux kernel network stack. The transport layer parameters are defined in the `rconn` C structure (struct) inside the OVS code. There is one `rconn` C structure per transport connection between the controller and the switch. For instance, the OpenFlow and OVSDB connections use different `rconn` C structures, even though the endpoints are the same. The `rconn` C structure is used to maintain socket, port, and other protocol-related information. To support quicSDN, the `rconn` C structure needs to be modified to support UDP. The OpenFlow and OVSDB protocols are implemented as service objects inside the OVS code. Each service object is an abstract protocol process. For instance, in order to start the OpenFlow and OVSDB services, OVS creates a service object for each service, and each service object is tied to its `rconn` C structure.

The OpenFlow and OVSDB services in OVS are created by issuing the CLI commands presented in §III-B. While an OVS service is created, the `vconn_lookup_class()` function looks up the requested transport protocol in the CLI against a list of predefined connection classes. The UDP connection class `udp_vconn_class` inherits the connection-related function pointers, including `open()`, `close()`, `connect()`, `recv()`, and `send()`. The `open()` function establishes a connection with the controller and must not be blocked while waiting for the connection requests or responses. If connection establishment does not complete immediately, the socket returns `EINPROGRESS` and retries in the background. `close()` tears down the connection gracefully,

`send()` sends, and `recv()` receives OpenFlow messages. Similar to `open()`, `recv()` does not block while waiting for the messages to arrive.

Using the newly modified transport layer infrastructure, the function `new_udp_uds()` opens a UDP socket for each OVS service. These sockets are registered to the new FDs in function `new_uds_fd()`, which are attached to their function pointers `open()`, `close()`, `recv()`, and `send()`. At this point, the `rconn` C structures are populated and the OVS services enter into the CONNECTING state. The OpenFlow and OVSDB services' state are dependent on the underlying transport layer protocol. Since there are no state transitions in UDP to show whether the connection is in an established state or not, the OVS services immediately transitions into the ACTIVE state.

B. QUIC Client and Server

QUIC uses a client-server model. The original development goal of QUIC was to replace TCP as a reliable transport protocol in HTTP3; however, the quicSDN architecture is different from HTTP3. Specifically, unlike HTTP3, multiple applications interact with QUIC in the quicSDN architecture, and these applications communicate with each end-point over the same UDP connection. In quicSDN's switch side implementation, `ovsdb-server` and `ovs-switchd` communicate with `quic-client`. On the controller side, `ryu-of` and `ryu-ovsdb` communicate with `quic-server`. We picked `ngtcp2` [48] for QUIC code, because it is updated frequently with bug fixes. `ngtcp2` also provides easiness in feature extension in OVS (as it is written in C) and is convenient for any feature integration with RYU code (via Python C extensions) on the controller side.

We divided `quic-server` and `quic-client` into server-agent, client-agent, and common APIs. The server-agent APIs are responsible for serving requests from clients, invoking common APIs, negotiating versions, and completing QUIC handshakes. The client-agent APIs are implemented to prepare the requests, rearrange the responses, and interact with common APIs. The common APIs are responsible for invoking the zero RTT scenario, encrypting and decrypting packets, and storing the cryptographic keys. This section presents the modifications relevant to quicSDN.

1) *QUIC Client:* Algorithms 1 and 2 present the pseudo-code of `quic-client` module. `quic-client` spawns two UDP servers listening on ports 6653 and 6640 on localhost (A1: L5-6)⁴ to intercept all connection requests and data packets from `ovsdb-server` and `ovs-switchd`. There are three sockets in `quic-client`: two sockets for the above-mentioned UDP servers, and one socket for connecting to `quic-server`. For these three sockets, we define three C structures to store connection information such as IP address, port, and socket information. These three C structures are for `ovs-switchd` (struct `p_openflow`), `ovsdb-server` (struct `p_ovsdb`), and `quic-server` (struct `p_quic`).

`ngtcp2` allows only one IP address and port to be specified in the CLI commands. We developed a new CLI command to

⁴This notation means Algorithm 1, Lines 5 through 6.

Algorithm 1: Pseudo-code of quic-client

```

1 function main ()
2   p_openflow, p_ovsdb, p_quic = {sock, port, addr}
3   client_arg = {p_openflow, p_ovsdb, p_quic}
4   populate client_arg from CLI
5   p_openflow = Connect to ovs-switchd on port 6653
6   p_ovsdb = Connect to ovsdb-server on port 6640
7   if !(start_client(client_arg) then
8     | return failure
9   close(p_openflow, p_ovsdb, p_quic)
10  return
11 function start_client (client_arg) ()
12  s1 = client_arg→p_openflow→sock
13  File *fp_ofl = fileno(s1)
14  s2 = client_arg→p_ovsdb→sock
15  File *fp_odb = fileno(s2)
16  sock = create UDP socket to connect to quic-server
17  File *fd_ = fileno(sock)
18  // set event callbacks
19  fd_ → readcd(), writecb()
20  fp_ofl → ofl_cb()
21  fp_odb → odb_cb()
22  _quic = init()
23  if !(_quic → run(client_arg) then
24    | return failure
25  return
26 function init ()
27  _quic→client = Initialize new client
28  set fd_, fp_ofl, fp_odb event callbacks in _quic
29  return _quic
30 function run (client_arg)
31  if (session_file) then
32    // 0-RTT Scenario
33    if !(resume()) then
34      | return failure
35  else
36    // 1-RTT Scenario
37    do_handshake()
38    if !(connect() then
39      | return failure
40  // Starting event loop
41  ev_run(ev_d, 0)
42  return
43 function readcb (ev_loop *loop, ev_io *w)
44  auto c = <client *w→data;
45  on_read()
46 function writecb (ev_loop *loop, ev_io *w)
47  auto c = <client *w→data
48  on_write()
49 function ofl_cb (ev_loop *loop, ev_io *w)
50  auto c = <ofl *w→data
51  this→type_flag = openflow
52  on_ofl_odb_read()
53 function odb_cb (ev_loop *loop, ev_io *w)
54  auto c = <ofdb *w→data
55  this→type_flag = ovsdb
56  on_ofl_odb_read();

```

populate the above-mentioned three C structures (A1: L3) on the quic-client side:

```
$ <quic_client_path> <quic server addr> <quic server
port> <openflow port> <ovsdb port>
```

To support asynchronous I/O operations, each socket is mapped to a stream pointer FD. In conjunction with existing FD (A1: L17) for a socket connected to quic-server, two more FDs named fp_ofl (A1: L12-13) and fp_odb (A1: L14-15) are introduced for each socket connected to ovs-switchd

Algorithm 2: Pseudo-code of quic-client (continued from Algorithm 1)

```

1 function on_read ()
2   array<uint8_t, 65536> buf
3   while true do
4     if !(recvfrom(this→fd_, buf.data, buf.len)) then
5       | return failure
6     if !(feed_data(buf.data, buf.len) then
7       | return failure
8 function feed_data (uint8_t data, int data_len)
9   if handshake_completed then
10    if !_con_rcv(data, datalen, &stream_id) then
11      | return failure
12    if stream_id is divisible by 3 then
13      | sendto(this → fp_ofl)
14    else
15      | sendto(this → fp_odb)
16  else
17    if !do_handshake(data, datalen) then
18      | return failure
19    else
20      | handshake_completed = true
21  return
22 function on_write ()
23  if !handshake_completed then
24    if !do_handshake(data, data_len) then
25      | return failure
26    else
27      | handshake_completed = true
28  while true do
29    if send_queue.size() <= 0 then
30      | break
31    buf = send_queue.front()
32    pkt_buf, error = _conn_write_pkt(buf)
33    if error != null then
34      | return failure
35    write_streams(pkt_buf)
36    return
37 function write_streams (buf)
38  if (this→openflow) then
39    int stream_id =
40      generate_stream_id_divisibile_by_3()
41  else if (this→ovsdb) then
42    int stream_id = generate_normal_stream_id()
43  on_write_stream(stream_id, buf)
44  return
45 function on_write_stream (stream_id, buf)
46  while true do
47    auto n = _conn_write_stream(ndatalen)
48    if n > 0 && and ndatalen > 0 then
49      | data.seek(ndatalen)
50      | send_packet()
51    if buf.size() = 0 then
52      | break
53  return
54 function on_ofl_odb_read ()
55  array<uint8_t, 65536> buf_ofl
56  array<uint8_t, 65536> buf_odb
57  if activity_detected of fp_ofl then
58    if (recvfrom(this→fp_ofl, buf_ofl.data, buf_ofl.len)) then
59      | send_queue.push(buf_ofl)
60  if activity_detected of fp_odb then
61    if (recvfrom(this→fp_odb, buf_odb.data, buf_odb.len)) then
62      | send_queue.push(buf_odb)
63  return

```

and ovsdb-server, respectively. Any activity detected on these

FDs invokes a callback function. We developed two new callback functions, `ofl_cb()` and `odb_cb()` (A1: L46-53) for `ovsdb-server` and `ovs-switchd` and modified the existing ones, `readcb()` (A1: line 40) and `writecb()` (A1: L43). `readcb()` is invoked when the FD receives a packet. Inside `readcb()`, `on_read()` receives data from the socket and passes it to `feed_data()` (A2: L8-20), which is responsible for processing QUIC handshake and data packets. If the QUIC handshake has been completed successfully, then it is confirmed that all the necessary security keys are in place (A2: L23-27). The function `_con_recv()` checks if the received packet contains the long or short QUIC header by inspecting the most significant bit of octet 0 (0x80) (A5: L1-6). The long header is used for QUIC version [49] and 1-RTT keys negotiations, and the short header is used for subsequent data communications. `crypt_quic_message()` (IV-B3) (A5: L6) parses the packet and performs all the necessary QUIC related operations such as encrypting packets, decrypting packets, and key management.

`writecb()` is invoked to send the packet to `quic-server`. The QUIC handshake is initiated by the `on_write()` function (A2: L23). Inside `on_write()`, `_conn_write_pkt()` encrypts the packet (A5: 10-12). `write_streams()` is then called to check if the packet is destined for `ovs-switchd` or `ovsdb-server` in order to generate appropriate stream IDs (A2: L38-41).

Packets that are received on the sockets connected to `ovsdb-server` and `ovs-switchd` invoke `odb_cb()` and `ofl_cb()` callbacks (A1: L46-53), respectively. Both callbacks invoke `on_ofl_odb_read()`, where packets are pushed to the `send_queue()` for QUIC processing (A2: L54-61).

2) *QUIC Server*: Algorithms 3 and 4 present the `quic-server` implementation. `quic-server` connects to `ryu-of` and `ryu-ovsdb` modules (A3: L5-6), which are listening on ports 6653 and 6640, respectively. Similar to `quic-client`, there are three sockets in `quic-server`. Two sockets are for `ryu-of` and `ryu-ovsdb` for port 6653 and 6640, respectively, while one socket is for a connection to `quic-client`. In order to store the connection information of these three sockets, we define three C structures, `p_quic`, `p_openflow` and `p_ovsdb` for `quic-client`, `ryu-of`, and `ryu-ovsdb` respectively. As previously mentioned, `ngtcp2`'s CLI commands contain only one IP address and port; therefore, we developed a new CLI command for `quic-server` to populate the three C structs with the appropriate information (A3: L3):

```
$ <quic_server_path> <quic server addr> <quic server
  port> <key> <certificate> <openflow port> <ovsdb
  port>
```

The above-mentioned three sockets in `quic-server` are capable of asynchronous I/O operations (A3: L14). Three FDs are mapped as stream pointers to these three sockets. Among the three FDs, the existing FD (`fd_`) is modified, and two new FDs, `fp_ofl` and `fp_odb` (A:3 L15-16), are added. The FD `fd_` is for the QUIC Connection to `quic-client`, FD `fp_ofl` is for the connection to `ryu-of`, and FD `fp_odb` is for the connection to `ryu-ovsdb`. These FDs monitor the sockets via an event loop and invoke callbacks if any activity is detected. Callbacks `readcb()` and `writecb()` are invoked if activity

Algorithm 3: Pseudo-code of `quic-server`

```
1 function main()
2   p_openflow, p_ovsdb, p_quic = {sock, port, addr}
3   server_arg = {key, cert, p_ovsdb, p_openflow, p_quic}
4   Populate server_arg from CLI
5   p_openflow = Connect to ryu-of on 6653 port
6   p_ovsdb = Connect to ryu-ovsdb on 6640 port
7   if !(start_server(server_arg) then
8     | return;
9   close(p_openflow, p_ovsdb, p_quic)]
10  return;
11 function start_server(server_arg)
12  s1 = server_arg→p_openflow→sock
13  s2 = server_arg→p_ovsdb→sock
14  sock = create UDP server to accept quic-client connections
15  File *fp_ofl = fileno(s1)
16  File *fp_odb = fileno(s2)
17  File *fd_ = fileno(sock)
18  // set event callbacks
19  fd_ → readcb(), writecb()
20  fp_ofl → ofl_cb()
21  fp_odb → odb_cb()
22  ev_run(ev_d, 0)
23 function readcb(ev_loop *loop, ev_io *w)
24  auto c = <client *>w→data
25  on_read()
26 function writecb(ev_loop *loop, ev_io *w)
27  auto c = <client *>w→data
28  on_write()
29 function ofl_cb(ev_loop *loop, ev_io *w)
30  auto c = <ofl *>w→data
31  this→type_flag = openflow
32  on_ofl_odb_read()
33 function odb_cb(ev_loop *loop, ev_io *w)
34  auto c = <ofdb *>w→data
35  this→type_flag = ovsdb
36  on_ofl_odb_read()
```

is detected on `fd_`. Similarly, callback `ofl_cb()` (A:3 L29-32) is invoked if any activity is detected on `fp_ofl`, and callback `odb_cb()` (A:3 L33-36) is invoked if any activity is detected on `fp_odb`.

The `on_read()` function is responsible for reading `fd_` to process the received QUIC packets (A:4 L1-12). First, the received QUIC packet is evaluated to check if the header in the corresponding packet is a long header or a short header (A:4 L7). Then the packet is passed to the `_accept()` function for decryption (A:4 L11).

The `on_write()` function is for sending packets to `quic-client` (A:4 L13-29). This function first evaluates and performs a QUIC handshake with `quic-client` to exchange cryptographic keys (A:4 L15). The buffer received in `on_write()` contains the OpenFlow or OVSDDB port information to maintain an external map (`_conn_map`) of port to streamID mapping for reverse traffic (A:4 L21). The function `on_write_stream()` searches for an existing stream, and if the stream does not exist yet, it opens a new stream and packs the data into it (A:4 L22). `_conn_write_pkt()` is responsible for encrypting the packets and placing them into the transmission queue (A:5 L10-12).

The function `on_ofl_odb_read()` is called by `ofl_cb()` and `odb_cb()` callbacks (A:4 L30-44). This function is responsible for exchanging packets between

Algorithm 4: Pseudo-code of quic-server (continued from Algorithm 3)

```

1 function on_read()
2   buffer<uint8_t, int, port> buf
3   while true do
4     if !(recvfrom(this→fd, buf.data, buf.len)) then
5       return
6     hd = this→hd
7     if (buf[0] & 0x80) then
8       | _pkt_decode_hd_long(&hd, buf.data())
9     else
10      | _pkt_decode_hd_short(&hd, buf.data())
11      | _accept(buf.data(), buf.len)
12  return
13 function on_write()
14  if !(handshake_completed) then
15    do_handshake()
16    handshake_completed = true
17  else
18    if !schedule_retransmit() then
19      return failure
20  buf = send_queue.front()
21  stream_id = _conn_map[buf→port]
22  on_write_stream(stream_id)
23  for ;; do
24    n = _conn_write_pkt()
25    if n = 0 then
26      break
27    buf_.push(n);
28    send_packet(buf)
29  return
30 function on_ofl_odb_read()
31  while true do
32    buffer<uint8_t, int, port> buf_ofl
33    if (recvfrom(this→fp_ofl, data, datalen)) then
34      buf_ofl.data = data
35      buf_ofl.len = datalen
36      buf_ofl.port = this→port;
37      // e.g 6653 for openflow
38      send_queue.push(buf_ofl)
39    buffer<uint8_t, int, port> buf_odb;
40    if (recvfrom(this→fp_odb, data, datalen)) then
41      buf_odb.data = data
42      buf_odb.len = datalen
43      buf_odb.port = this→port
44      // e.g 6640 for ovsdb
45      send_queue.push(buf_odb)
46  return

```

ryu-of, ryu-ovsdb, and quic-server.

3) `crypt_quic_message()`: This API consists of the `decrypts_message()` and `encrypts_message()` functions, responsible for decrypting and encrypting packets in multiple phases. Each phase has a different set of keys. Before acquiring the symmetric keys, QUIC completes four phases. The first phase is the Initial Key Agreement, where each party sets and exchanges the initial key and additional information, such as HMAC. Both parties then agree to a common key (ik), which is derived from the Client Initial Key (ik_c) and the Server Initial Key (ik_s). The second stage is the Initial Data Exchange, where data is encrypted and decrypted by using an Authenticated Encryption with Associated Data (AEAD) Scheme [50] and ik . The third phase is the Key Agreement, where the session key (k) is derived from the client session key (k_c), server session key (k_s), and aux , where $aux \in \{0, 1\}$.

Algorithm 5: Common APIs

```

1 function _con_recv(data, datalen, &s)
2   if (data[0] & 0x80) then
3     | _pkt_decode_hd_long()
4   else
5     | _pkt_decode_hd_short()
6   crypt_quic_message(decrypt)
7 function _conn_write_stream()
8   find_stream_info(dest)
9   _conn_write_pkt()
10 function _conn_write_pkt()
11  conn_write_probe_pkt()
12  crypt_quic_message(encrypt)
13 function _accept(data, datalen)
14  plain_text = crypt_quic_message(decrypt)
15  if (this→stream_id is divisible by 3) then
16    | _conn_map[opnflow_port] = stream_id
17    | sendto(this→fp_ofl)
18  else
19    | _conn_map[odb_port] = stream_id
20    | sendto(this→fp_odb)

```

The fourth phase is the Data Exchange. In this phase, data is sent using the associated AEAD scheme and k . The server uses k_c to encrypt and k_s to decrypt packets, while the client uses k_s to encrypt and k_c to decrypt packets. In addition, `crypt_quic_message()` also prepares the initialization vector (iv) and salt for the cryptographic keys.

C. RYU

In tcpSDN, RYU inherits the TCP transport layer infrastructure in the form of base classes. The most important base class is `OFPHandler`, which declares a controller base class object called `OpenFlowController`. Inside this object, a server is spawned to receive and process all the received packets via an event loop. Any packet received will be pushed to the RYU app for processing. In quicSDN, to make RYU compatible with UDP, the first task is to replace the TCP infrastructure and have RYU spawn a UDP server instead. This modification is challenging due to RYU's current event loop callback mechanism. This callback mechanism is based on asynchronous TCP socket, which is a part of eventlet I/O library. In order to make it UDP based, first the eventlet library needs to be changed to support UDP sockets. Moreover, RYU needs to implement the server based on the eventlet library UDP sockets. In quicSDN, the RYU applications `ryu-of` and `ryu-ovsdb` are started using the following CLIs commands:

```

$ ryu-manager -ofp-listen-port <port num> <app name>
$ ryu-manager <OVSDDB app name>

```

Note that no changes were made to the state machines of `ryu-of` and `ryu-ovsdb`. In tcpSDN, packets are processed in the eventlet library which is implemented using `GreenSocket`. We modified the RYU event loop to make sure that packets are directly pushed from eventlet library to be processed in the app itself. The OpenFlow (`ryu-of`) and OVSDB (`ryu-ovsdb`) controllers both use the same transport layer infrastructure.

V. EMPIRICAL EVALUATION

In this section, we empirically evaluate quicSDN versus tcpSDN. The testbed configuration is demonstrated in Figure

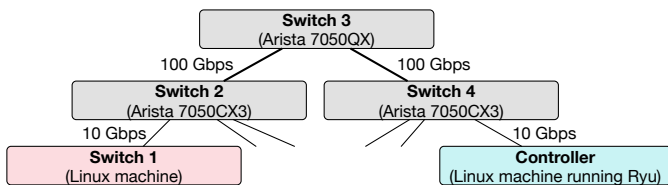


Fig. 7: Testbed topology. Switch 1 and the Controller communicate through Switch 2, 3, and 4.

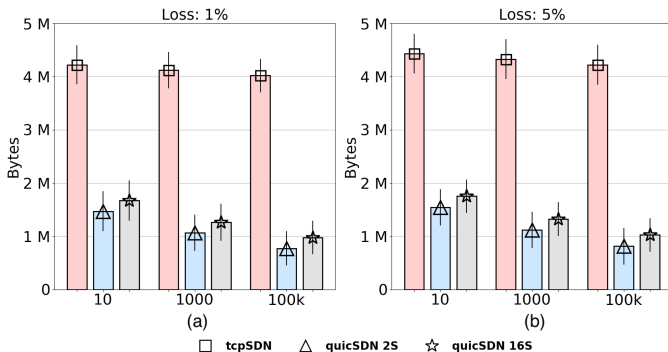


Fig. 8: The protocol overhead of tcpSDN and quicSDN in the *short RTT* scenario. The x-axis shows three message generation rates: 10, 1000, 100k messages per sec. Sub-figures (a) and (b) present the overhead for 1% and 5% packet loss rates, respectively.

7. We use three high-performance switches: one 7050QX (Switch 3) and two 7050CX3 (Switch 2 and 4). For Switch 1, we use a Linux machine, which allows us to run OpenFlow and OVSDB, as well as emulating the execution of more agents on the switch to generate various traffic patterns. The RTT between Switch 1 and the Controller shows a mean value of 0.32 ms and a standard deviation of 0.051 ms; we refer to this scenario as *short RTT*. We also configured the testbed to generate considerably higher RTT values with a mean value of 150 ms and a standard deviation of 50 ms; we refer to this scenario as *long RTT*. This long RTT represents a scenario where the data plane is controlled by a remote controller or when there is a considerable traffic competing with southbound control traffic. For each experiment performed, the total number of messages generated from Switch 1 to the Controller and vice-versa are 50,000 on each side. Also, we vary the message generation rate per seconds as 10, 1000, 10,000 messages per second. To represent various congestion levels in the switches, we introduce packet loss rates of 1% and 5%. TCP Segmentation Offload (TSO) and Generic Receive Offload (GRO) have been disabled on Switch 1 and the Controller.

We consider two variants of quicSDN and compare them against tcpSDN. These variants are explained as follows. **quicSDN-2s**: In this implementation, all the messages generated by OpenFlow are assigned to stream 0 and all the OVSDB generated messages are assigned to stream 2. Note that these streams are bidirectional; for example, stream ID 0 is used for bidirectional OpenFlow messages between Switch 1 and the Controller. **quicSDN-16s**: This implementation opens 16 bidirectional streams. As explained in Section III-B, each message is assigned a different stream identifier in the range 0 to 15, in a round-robin fashion.

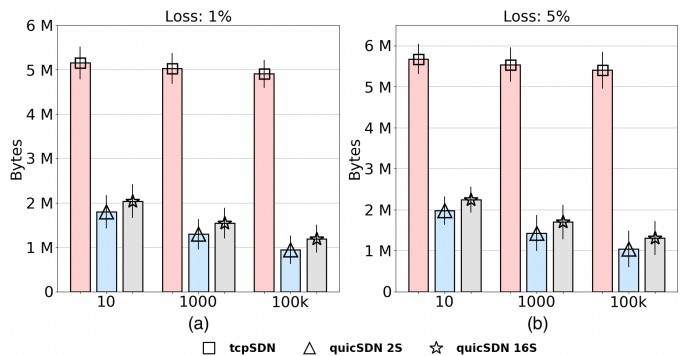


Fig. 9: The protocol overhead of tcpSDN and quicSDN in the *long RTT* scenario. The x-axis shows three message generation rates: 10, 1000, 100k messages per sec. Sub-figures (a) and (b) present the overhead for 1% and 5% packet loss rates, respectively.

A. Overhead

We first evaluate the communication overhead between Switch 1 and the Controller for exchanging OpenFlow and OVSDB messages. Whenever a controller sends a message (command) to the switch, the Controller needs to receive a reply from the switch to ensure proper enforcement of the commands. Therefore, we measure the *bidirectional* communication overhead between the two nodes. Figures 8 and 9 present protocol overhead for the short-RTT and long-RTT scenarios, respectively. The results show that the overhead of quicSDN is considerably lower than tcpSDN. As we increase the number of messages exchanged per second between the switch and the controller, the communication overhead of quicSDN and tcpSDN drops; also, the decline of quicSDN overhead is higher than tcpSDN. These empirical results conform to the analytical results we presented in Section II-A. The lower overhead of quicSDN is due to two main reasons: *First*, multiple agents (OpenFlow and OVSDB in these experiments) use a single connection to communicate with the controller. Therefore, by multiplexing the messages of these two agents, quicSDN achieves a lower probability of sending packets smaller than MTU, compared to tcpSDN. *Second*, the transport header overhead of quicSDN is lower than tcpSDN when the number of streams is less than ten. As we increase the number of streams from two to ten, the number of QUIC frames per packet may increase as well, depending on the message size. Although a higher number of streams results in a better mitigation of the HOL problem, this benefit comes at the cost of higher transport layer overhead. Nevertheless, as the results show, the overhead of quicSDN-16s is still lower than tcpSDN.

Figures 8 and 9 show that the overhead of tcpSDN and quicSDN are higher in the long RTT scenario compared to the short RTT scenario. This overhead is caused by the fluctuations in RTT, which cause out of order packets and more number of retransmissions. However, the overhead increase of quicSDN is lower than that of tcpSDN. For example, for 1% loss rate and 10 messages/second, increasing the RTT causes the overhead of tcpSDN to increase by 25.9%, while the overhead of quicSDN-2s increases by 19.5%. TCP does not support NACK, instead it supports Selective Acknowl-

edgement (SACK)⁵. While NACK specifies the packets that a receiver has *not* received, SACK specifies the range of packets a receiver has received. These notifications are specified as ranges. TCP allows including up to three SACK blocks [51], whereas, QUIC supports 256 NACK blocks [32]. This makes a difference when packet loss is eminent, as supporting only three blocks cause more number of ACKs with SACK options than the number of ACKs with NACK.

B. Message Delivery Delay

In this experiment, we measure message delivery delay between the Controller and Switch 1 (in Figure 7). We define message delivery delay as the time interval between the time instance a message is generated by an application running on the sender until complete message reception by the application running on the receiver. We use two message size pairs: $\langle 400, 600 \rangle$, and $\langle 3000, 600 \rangle$. Each pair represents the size of messages generated by the sender’s application. We also vary RTT and packet loss rate between the two nodes.

Figure 10 shows the result. First, we observe that the message delivery delay of tcpSDN is higher than quicSDN-2s and quicSDN-16s. The primary reason behind this enhancement is that, in contrast with tcpSDN, quicSDN can immediately deliver a stream payload to the application if it contains an entire message, regardless of the loss of packets with smaller sequence number. Second, the message delivery delay of quicSDN-2s is higher than quicSDN-16s. This is because, as the number of streams increases, QUIC can determine the independence between frames (inside packets) more effectively. The final observation is that for both quicSDN-2s and quicSDN-16s, message delivery delay is more fluctuating than tcpSDN, and this is due to the order of packet arrivals and interdependence between frames included in each packet. In the following, we present more details regarding the underlying causes of the higher performance of quicSDN compared to tcpSDN.

Assume an application generates message sizes 400 and 600 bytes, simply denoted as $\langle 400, 600 \rangle$. Figure 11a presents a sample packet transmission from a sender to a receiver. Assume each packet can include 1470 bytes of message data. At time t_1 , the sender generates and includes four messages in Packet i : M1 (400 bytes), M2 (600 bytes), M3 (400 bytes), and part of M4 (600 bytes). Packet i includes 100 bytes of M4, and the remaining 500 bytes of this message are included in Packet $i+1$. Packet i , which is transmitted at time t_1 , is lost along the path between the sender and receiver. Packet $i+1$ transmitted at time t_2 is fully received at t_4 . Note the transmission time of each packet is L/r , where L is packet size and r is the link speed between the two nodes. Propagation delay between the two nodes is $RTT/2$. At t_4 , in a quicSDN network, since messages M5 and M6 are fully received, they are delivered to the application, without having to wait for Packet i . In contrast, in a tcpSDN network, since the receiver needs to receive all the bytes in order, it needs to wait for Packet i before processing Packet $i+1$. Assuming the RTO of the sender is RTT , Packet

i is retransmitted at t_5 .⁶ At t_7 , Packet i is fully received and tcpSDN delivers all the sent messages to the application, while quicSDN delivers the remaining messages only, i.e., M1, M2, M3, and M4. In this example, for quicSDN, the delivery delay of M5 and M6 is $RTT/2 + L/r$. With tcpSDN, the delivery delay of these two messages is $RTT + RTT/2 + 2L/r$. In a realistic scenario, for example, if M5 and M6 are two flow rules, the switch can process and install these rules with a shorter delay compared to tcpSDN, thereby resulting in faster reaction to network dynamics.

Figure 11b presents a sample packet transmission from a sender to a receiver where the message sizes generated by the sender are $\langle 3000, 600 \rangle$. At time t_1 , the sender includes the constituting bytes of message M1 (3000 bytes) in Packet i , Packet $i+1$, and Packet $i+2$. Packet $i+2$ includes the remaining bytes of messages M1 (60 bytes), message M2 (600 bytes), and some bytes of message M3 (810 bytes). Packets $i+1$ and $i+2$ transmitted at t_1 and t_2 are lost. At time t_5 , packet $i+1$ is fully received. At this time, in a quicSDN network, M2 is delivered to the application. In a tcpSDN network, since the delivery of M2 is contingent upon the successful reception of all these three packets, messages M1, M2 and M3 are delivered to the application at time t_9 . This scenario justifies the effect of increasing message size on message delivery delay and conforms with the empirical results presented in Figure 10 that show delivery delay is higher for larger message sizes.

VI. RELATED WORK

In this section, we first review the widely-used methods to enhance the scalability of SDNs and then review the existing studies on the performance and applications of QUIC protocol.

A. SDN Scalability

The communication overhead and delay between a controller and its associated switches have been explored in the literature. The primary methods used to mitigate these overheads are: (i) increasing each switch’s autonomy to handle flows, (ii) selecting optimal controller placement, and (iii) using multiple controllers to reduce switch-controller distances.

To reduce the amount of switch-controller communication, Hedera [52] allows switches to handle mice flows via Equal-Cost Multi-Path (ECMP), and switches only consult the controller when dealing with elephant flows. DIFANE [53] distributes OpenFlow wildcards across switches to allow them to perform local routing. Curtis et al. [17] show that polling statistical data from switches reduces flow rule setup rate. They also demonstrate that the low bandwidth between a switching appliance’s CPU and ASIC introduces a significant communication delay between switch and controller when installing new flow rules. They propose DevoFlow, which devolves the control of many flows back to switches, and the controller only targets significant flows. DevoFlow uses wildcard rules to reduce the number of interactions with the controller, while also reducing the usage of Ternary Content

⁵This feature must be enabled as a socket option.

⁶RTO value is usually higher than RTT to account for RTT variations.

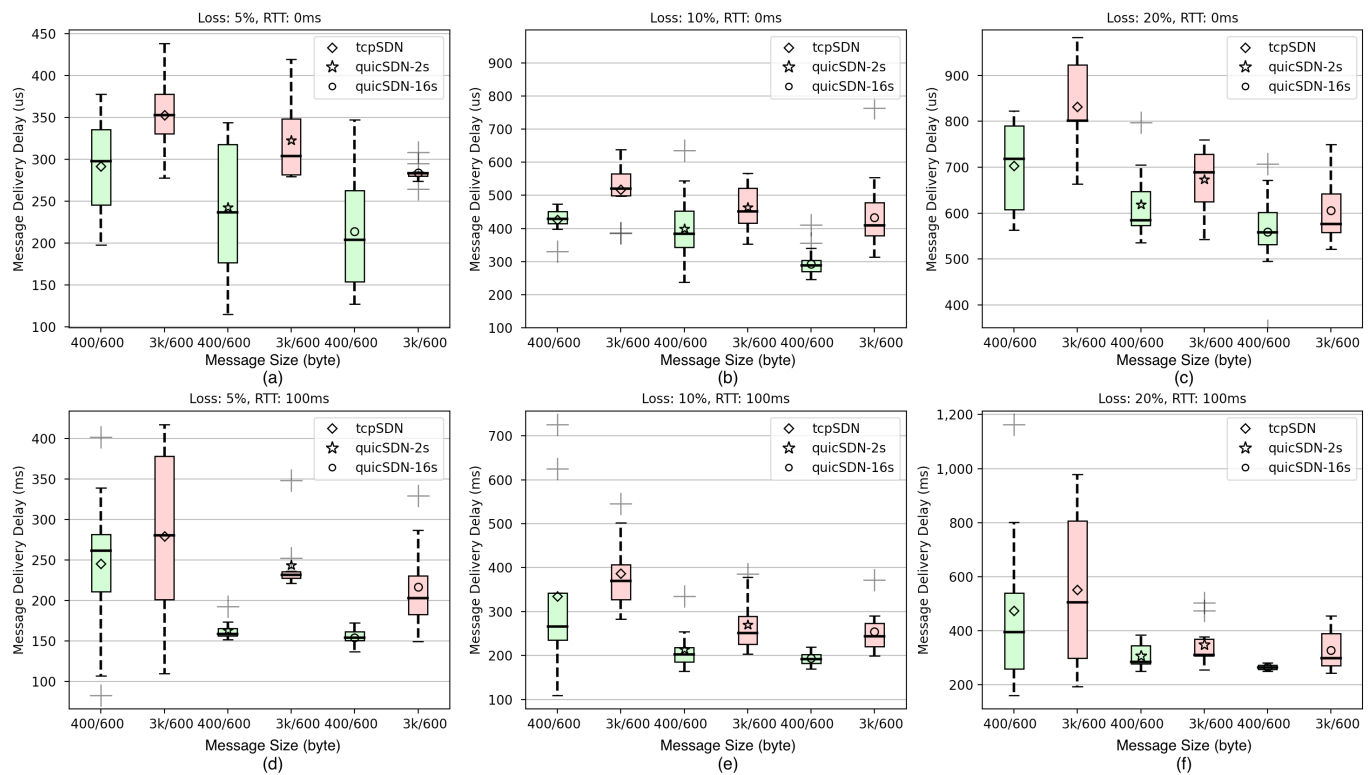


Fig. 10: Message delivery delay. The experiment is performed 30 times for each configuration. The results confirm the lower message delivery delay of quicSDN compared to tcpSDN. Also, quicSDN-16s provides shorter delivery delay compared to quicSDN-2s because using a higher number of streams increases the effectiveness of determining independence among packet frames.

Addressable Memory (TCAM). Mahout [54] uses sender’s TCP buffer size to identify mice flows and decide whether communication with the controller is necessary or not. Kim et al. [55] propose a flow management scheme to reduce the number of OpenFlow Packet_in messages sent to the controller, thereby reducing the network overhead caused by entry misses in a flow table. Their proposed scheme reduces table misses by maintaining inactive flow entries for as long as possible. The inactive flow entries are maintained as long as the flow table has space; inactive flow entries are deleted once the flow table starts filling up. Qin et al. [42] analyzed controller-switch and inter-controller traffic overheads in networks with varying numbers of nodes. They show that the relationship between the amount of control traffic and the number of nodes in a network is linear. They model and propose a solution to the controller placement problem, reducing device management delay by 25%. Van Bemten et al. [56] use switches from multiple vendors and demonstrated that switch management operations are not predictable and reliable. For example, with Pica switches, as the number of Flow_mod messages per second increase, the switch shows two behaviors: the number of ignored rules increases, and some rules are reported to be installed while they have not been.

Onix [57] provides a wide range of primitives for developing control applications in environments such as WAN and public clouds. To simplify this process while maintaining scalability, APIs are provided for a distributed implementation. For example, control applications can utilize these APIs to access the information maintained by Onix instances. HyperFlow [58]

synchronizes the status of distributed controllers and provides control applications with uniform, consistent access to the overall network data. Kandoo [59] assumes local processing is available close to the switches. Applications that rely on local information are assigned to the local controllers, while non-local applications run in a root controller. Bera et al. [39] propose a dynamic controller assignment scheme to maximize controller reactivity in heterogeneous networks. They accomplish this by selecting a controller to manage new flows that arrive at switches in the network, such that controller-switch delay and protocol overheads are optimized. Disco [60] targets synchronization among controllers that manage multiple, heterogeneous networks. They use Advanced Message Queuing Protocol (AMQP), which utilizes TCP, to support east-west communication among controllers; AMQP allows controllers to subscribe and publish to topics.

Zhang et al. [61] propose a min-cut algorithm for controller placement to enhance communication reliability with controllers. The network is first partitioned with the minimum inter-partition cut, and inside each partition, the node with minimum distance to other nodes is found. Survivor [40] uses path diversity as a metric of their formulated linear programming model to determine controller location. Simulation results show that the connectivity loss of the proposed method is between 2 to 3x less than [61]. Beheshti et al. [62] argue the importance of providing switches with alternative paths to connect to the controller as soon as the primary path is dropped. The proposed routing algorithm takes into account both distance and resilience to path failures.

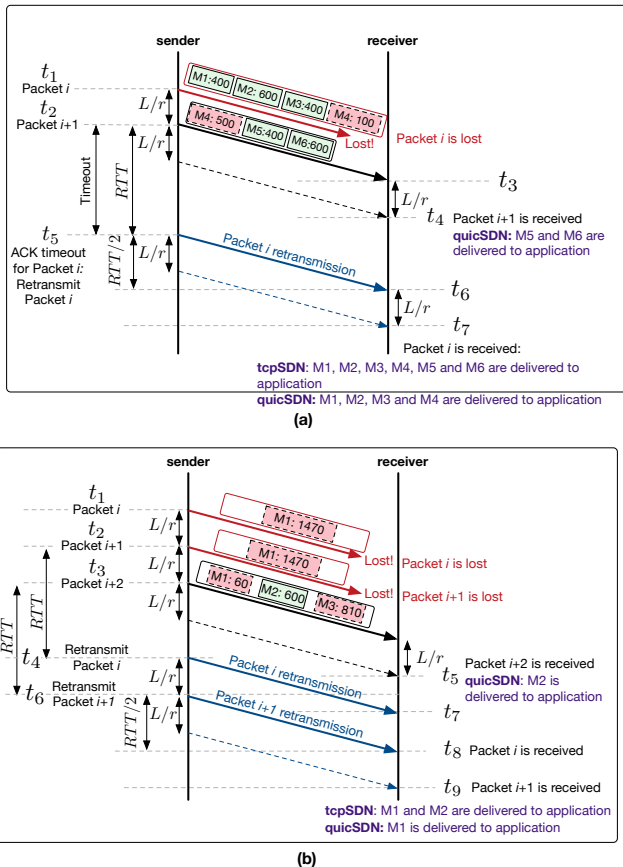


Fig. 11: The effect of packet loss and message size on the message delivery delay of tcpSDN and quicSDN. Here, L is the packet length (bits) and r is the transmission rate (bps) between the client and server. Red boxes show partial message inclusion in a packet and green boxes show complete inclusion of a message in a packet.

Despite the valuable insights these works provide into the scalability of SDNs, the effects of transport layer protocols on southbound communication have not been studied. The contributions of this paper are orthogonal to the existing works and can be leveraged to enhance SDN scalability further using methods such as dynamic switch-to-controller assignment and predictive flow installation.

B. QUIC Protocol

Started as an experimental protocol in Google, QUIC gained traction due to its benefits over TCP in several types of networks [63]–[65]. Existing studies present the performance benefits of QUIC over TCP when used for transferring messages of application protocols such as HTTP/1.1, HTTP/2, and HTTP/3. Zheng et al. [66] compared QUIC with TCP/TLS in HTTP/2 and show that QUIC performs better in congested networks where the retransmission rate is high. QUIC achieves a higher performance due to its packet pacing method, which reduces traffic burstiness and packet loss. Also, QUIC multiplexes several streams carrying data over one connection, which enables QUIC to carry more data than TCP during a single round trip. Multiplexing is specifically beneficial where medium access is expensive, such as on wireless networks. Biswal et al. [67] compared HTTP/3’s (QUIC) and HTTP/2’s

(TCP) page load times for large and small objects and showed that for 0% loss, HTTP/2 (TCP) performs better than HTTP/3 (QUIC). This is because with 0% loss, there is no congestion and the traffic flow is adjusted by flow control. Since QUIC employs flow control on each stream and the overall connection, this management of flow control per stream introduces latency. On the other side, in realistic scenarios where packet loss occurs, HTTP/3 (QUIC) outperforms HTTP/2 (TCP). This is because HTTP/2 is affected by the HOL blocking problem. Das et al. [68] compared HTTP/2 (SPDY), HTTP/1.1 (TCP) and HTTP/3 (QUIC) under different network parameters including page load time and packet loss. Their experiments were run for Amazon Alexa’s top 500 websites [69]. Their results show that HTTP/1.1 outperforms HTTP3 and HTTP2 for very minute bandwidth (0.2 Mbps). For a small bandwidth, the bandwidth-delay product ($bandwidth \times RTT$) is small, which means the amount of data carried in a round trip is small too. In HTTP/1.1, this translates into opening several TCP connections, while in HTTP/2 or HTTP/3, this causes opening several streams in one transport connection. Since HTTP/1.1 does not employ multiplexing, only the TCP connections experiencing packet loss are affected. On the other hand, HTTP/2 and HTTP/3 rely on a single transport connection with multiple streams; therefore, a single stream suffering from packet loss can affect the entire connection. When the bandwidth is increased to 1 Mbps, the bandwidth-delay product increases, packet loss rate drops, and HTTP/3 outperforms HTTP1.1.

VII. CONCLUSION

As the need for the higher rate, lower-overhead, and faster communication between the data plane and control plane in SDNs increases, the role of the transport protocol used by southbound protocols increases too. In this paper, we studied the shortcomings of using TCP and justified the benefits of QUIC over TCP in SDNs. We presented the design and implementation of quicSDN, a novel architecture that enables the communication of the control plane and data plane over the QUIC protocol. We presented the benefits of quicSDN over tcpSDN via analytical studies and empirical evaluations.

Some future work directions to enhance the quicSDN architecture are as follows. First, the proposed quicSDN architecture can be used to improve the efficiency of switch-to-controller assignment methods for purposes such as load balancing. Second, the proposed architecture can be leveraged for faster, more dynamic communication between switches and controllers to perform predictive configurations. Third, in the current implementation, we use UDSs for inter-process communication, which involves the Linux kernel’s network stack for processing the exchanged messages. The use and study of alternative methods are left as future work. For example, on end-devices such as servers running containers and VMs, the proposed architecture for switches may be integrated with Data Plane Development Kit (DPDK) to reduce inter-process communication overhead. Fourth, throughput and latency can be further improved by bringing Ethernet, IP, and UDP packet processing into userspace for QUIC packets. Fifth,

the benefits of QUIC's advanced flow and congestion control mechanisms have yet to be fully analyzed and quantified in SDNs.

ACKNOWLEDGMENT

This work has been partially supported by a gift fund from Arista Networks. Also, the authors would like to thank Arista Networks for donating the equipment used for this research.

REFERENCES

- [1] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.
- [2] F. Z. Yousaf, M. Bredel, S. Schaller, and F. Schneider, "Nfv and sdn—key technology enablers for 5g networks," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 11, pp. 2468–2478, 2017.
- [3] A. Kaloxylos, "A survey and an analysis of network slicing in 5g networks," *IEEE Communications Standards Magazine*, vol. 2, no. 1, pp. 60–65, 2018.
- [4] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing—a key technology towards 5g," *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.
- [5] C. Powell, C. Desiniotis, and B. Dezfouli, "The fog development kit: A platform for the development and management of fog systems," *IEEE Internet of Things Journal*, vol. 7, no. 4, pp. 3198–3213, 2020.
- [6] RYU Controller. (2017). [Online]. Available: https://ryu.readthedocs.io/en/latest/getting_started.html
- [7] OpenDayLight Controller. (2021). [Online]. Available: <https://www.opendaylight.org/>
- [8] J. Case, M. Fedor, M. L. Schoffstall, and J. Davin, "RFC1157: Simple network management protocol (SNMP)," 1990.
- [9] 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.
- [10] B. Pfaff and B. Davie, "The open vswitch database management protocol," *Internet Requests for Comments, RFC Editor, RFC*, vol. 7047, 2013.
- [11] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "Network configuration protocol (NETCONF)," 2011.
- [12] RYU. (2019) Arista Directflow SDK. [Online]. Available: <http://aristanetworks.github.io/EosSdk/docs/1.7.0/ref/directflow.html>
- [13] RYU. (2019) Cisco OpenFlow Plugin. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/switches/datacenter/sdn/configuration/b_openflow_agent_nxos/b_openflow_agent_nxos_chapter_01.html
- [14] RYU. (2019) HP OpenFlow. [Online]. Available: https://support.hpe.com/hpsc/public/docDisplay?docLocale=en_US&docId=emr_na-c04777809
- [15] M. Alsaeedi, M. M. Mohamad, and A. A. Al-Roubaiey, "Toward adaptive and scalable openflow-sdn flow control: A survey," *IEEE Access*, vol. 7, pp. 107 346–107 379, 2019.
- [16] M. Noormohammadpour and C. S. Raghavendra, "Datacenter traffic control: Understanding techniques and tradeoffs," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1492–1525, 2017.
- [17] 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, pp. 254–265.
- [18] S. Jouet, R. Cziva, and D. P. Pazaros, "Arbitrary packet matching in openflow," in *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2015, pp. 1–6.
- [19] R. Ying, W.-K. Jia, C. Luo, and Y. Wu, "Expedited eviction of invalid flow entries for sdn-based epc networks," in *IEEE/CIC International Conference on Communications in China (ICCC)*, 2019, pp. 298–303.
- [20] E.-D. Kim, Y. Choi, S.-I. Lee, and H. J. Kim, "Enhanced flow table management scheme with an lru-based caching algorithm for sdn," *IEEE Access*, vol. 5, pp. 25 555–25 564, 2017.
- [21] J. Chen, A. Gopal, and B. Dezfouli, "Modeling control traffic in software-defined networks," in *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. IEEE, 2021, pp. 258–262.
- [22] D. Palma, J. Goncalves, B. Sousa, L. Cordeiro, P. Simoes, S. Sharma, and D. Staessens, "The queuepusher: Enabling queue management in openflow," in *Third European workshop on software defined networks*. IEEE, 2014, pp. 125–126.
- [23] C. Caba and J. Soler, "APIs for QoS configuration in Software Defined Networks," in *Proceedings of the 1st IEEE Conference on Network Softwarization (NetSoft)*, 2015, pp. 1–5.
- [24] S. Sharma, D. Staessens, D. Colle, D. Palma, J. Goncalves, R. Figueiredo, D. Morris, M. Pickavet, and P. Demeester, "Implementing quality of service for the software defined networking enabled future internet," in *Third European workshop on software defined networks*. IEEE, 2014, pp. 49–54.
- [25] J. Flathagen, T. M. Mjelde, and O. I. Bentstuen, "A combined Network Access Control and QoS scheme for Software Defined Networks," in *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2018, pp. 1–6.
- [26] F. Volpato, M. P. Da Silva, A. L. Gonçalves, and M. A. R. Dantas, "An autonomic qos management architecture for software-defined networking environments," in *IEEE Symposium on Computers and Communications (ISCC)*, 2017, pp. 418–423.
- [27] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.
- [28] OpenConfig, "OpenConfig: Vendor-neutral, model-driven network management designed by users," 2016. [Online]. Available: <https://www.openconfig.net>
- [29] Arista. (2022) Arista Extensible Operating System (EOS), The World's Most Advanced Network Operating System. [Online]. Available: <https://www.arista.com/en/products/eos>
- [30] M. Karakus and A. Duresi, "A survey: Control plane scalability issues and approaches in software-defined networking (sdn)," *Computer Networks*, vol. 112, pp. 279–293, 2017.
- [31] J. Hu, C. Lin, X. Li, and J. Huang, "Scalability of control planes for software defined networks: Modeling and evaluation," in *IEEE 22nd International Symposium of Quality of Service (IWQoS)*, 2014, pp. 147–152.
- [32] J. Iyengar and M. Thompson. (2020) QUIC: A UDP-Based Multiplexed and Secure Transport. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc9000>
- [33] E. A. M. Bishop. (2019) Hypertext Transfer Protocol Version 3 (HTTP/3). [Online]. Available: <https://tools.ietf.org/html/draft-ietf-quic-http-32>
- [34] (2021) Eventlet Library. [Online]. Available: <https://eventlet.net/>
- [35] N. Mathewson, A. Khuzhin, and N. Provos. (2017) libevent—an event notification library. [Online]. Available: <https://libevent.org>
- [36] J. C. Mogul and G. Minshall, "Rethinking the tcp nagle algorithm," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 1, pp. 6–20, 2001.
- [37] T. Dierks and E. Rescorla, "The transport layer security (tls) protocol version 1.2," 2008.
- [38] E. Rescorla. (2018) The Transport Layer Security (TLS) Protocol Version 1.3. [Online]. Available: <https://tools.ietf.org/html/rfc8446>
- [39] S. Bera, S. Misra, and N. Saha, "Traffic-aware dynamic controller assignment in sdn," *IEEE Transactions on Communications*, vol. 68, no. 7, pp. 4375–4382, 2020.
- [40] L. F. Müller, R. R. Oliveira, M. C. Luizelli, L. P. Gaspary, and M. P. Barcellos, "Survivor: An enhanced controller placement strategy for improving sdn survivability," in *IEEE Global Communications Conference*, 2014, pp. 1909–1915.
- [41] A. Alowa and T. Fevens, "Combined degree-based with independent dominating set approach for controller placement problem in software defined networks," in *22nd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. IEEE, 2019, pp. 269–276.
- [42] Q. Qin, K. Poularakis, G. Iosifidis, and L. Tassiulas, "SDN Controller Placement at the Edge: Optimizing Delay and Overheads," *IEEE Conference on Computer Communications (INFOCOM)*, pp. 684–692, 2018.
- [43] J. Iyengar. (2016) QUIC at 10,000 feet. [Online]. Available: <https://docs.google.com/document/d/1gY9-YNDNAB1eip-RTPbqphgySwsNSDHLq9D5Bty4FSU/edit>
- [44] J. Iyengar. (2020) QUIC at 10,000 feet. [Online]. Available: <https://tools.ietf.org/html/draft-ietf-quic-recovery-33>
- [45] R. Odaira and K. Hiraki, "Selective optimization of locks by runtime statistics and just-in-time compilation," in *Proceedings International Parallel and Distributed Processing Symposium*. IEEE, 2003, pp. 6–pp.

- [46] H. Jung, H. Han, A. Fekete, G. Heiser, and H. Y. Yeom, "A scalable lock manager for multicores," *ACM Transactions on Database Systems (TODS)*, vol. 39, no. 4, pp. 1–29, 2014.
- [47] E. George. (2021) Eventlet open source project profile. [Online]. Available: <https://github.com/eventlet/eventlet/blob/master/eventlet/greenio/base.py>
- [48] (2021) ngtcp2 Project. [Online]. Available: <https://github.com/ngtcp2/ngtcp2>
- [49] D. Schinazi and E. Rescorla. (2020) QUIC Version Negotiation. [Online]. Available: <https://tools.ietf.org/html/draft-ietf-quic-version-negotiation-02>
- [50] P. Rogaway, "Authenticated-encryption with associated-data," in *Proceedings of the 9th ACM conference on Computer and communications security*, 2002, pp. 98–107.
- [51] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "Rfc2018: Tcp selective acknowledgement options," 1996.
- [52] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat *et al.*, "Hedera: dynamic flow scheduling for data center networks." in *NSDI*, vol. 10, no. 8, 2010, pp. 89–92.
- [53] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," *ACM SIGCOMM*, vol. 40, no. 4, pp. 351–362, 2010.
- [54] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *IEEE Conference on Computer Communications (INFOCOM)*, 2011, pp. 1629–1637.
- [55] E.-D. Kim, S.-I. Lee, Y. Choi, M.-K. Shin, and H.-J. Kim, "A flow entry management scheme for reducing controller overhead," *16th International Conference on Advanced Communication Technology*, 2014.
- [56] A. Van Bemten, N. Đerić, A. Varasteh, A. Blenk, S. Schmid, and W. Kellerer, "Empirical predictability study of sdn switches," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019, pp. 1–13.
- [57] 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.
- [58] 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*, vol. 3. USENIX, 2010.
- [59] 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*, 2012, pp. 19–24.
- [60] K. Phemius, M. Bouet, and J. Leguay, "Disco: Distributed multi-domain sdn controllers," in *IEEE Network Operations and Management Symposium (NOMS)*, 2014, pp. 1–4.
- [61] Y. Zhang, N. Beheshti, and M. Tatipamula, "On resilience of split-architecture networks," in *IEEE Global Telecommunications Conference (GLOBECOM)*, 2011, pp. 1–6.
- [62] N. Beheshti and Y. Zhang, "Fast failover for control traffic in software-defined networks," in *IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2012, pp. 2665–2670.
- [63] P. Kumar and B. Dezfouli, "Implementation and analysis of QUIC for MQTT," *Computer Networks*, vol. 150, pp. 28–45, 2019.
- [64] Y. Wang, K. Zhao, W. Li, J. Fraire, Z. Sun, and Y. Fang, "Performance evaluation of quic with bbr in satellite internet," in *2018 6th IEEE International Conference on Wireless for Space and Extreme Environments (WiSEE)*. IEEE, 2018, pp. 195–199.
- [65] R. Lychev, S. Jero, A. Boldyreva, and C. Nita-Rotaru, "How secure and quick is quic? provable security and performance analyses," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 214–231.
- [66] Y. Zheng, Y. Wang, M. Rui, A. Palade, S. Sheehan, and E. O. Nuallain, "Performance Evaluation of HTTP/2 over TLS+ TCP and HTTP/2 over QUIC in a Mobile Network," *Journal of Information Sciences and Computing Technologies*, vol. 7, no. 1, 2018.
- [67] P. Biswal and O. Gnawali, "Does quic make the web faster?" in *IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2016, pp. 1–6.
- [68] S. R. Das, "Evaluation of quic on web page performance," Ph.D. dissertation, Massachusetts Institute of Technology, 2014.
- [69] A. I. LLC. (2021) Alexa - Top Sites. [Online]. Available: <https://www.alexa.com/topsites>