Journal of Network and Computer Applications 222 (2024) 103780

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Research paper

quicSDN: Transitioning from TCP to QUIC for southbound communication in software-defined networks

Puneet Kumar, Behnam Dezfouli *

Internet of Things Research Lab, Department of Computer Science and Engineering, Santa Clara University, USA

ARTICLE INFO

Keywords:
Software-defined networks
Observability
Dynamic configuration
Overhead
Latency
Management
Agents

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 Transmission Control Protocol (TCP) to enforce reliability. 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 Quick UDP Internet Connection (QUIC) protocol can enhance the performance of this communication. We introduce the *quicSDN* architecture to enable 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. Our empirical evaluations in different settings demonstrate that quicSDN lowers communication overhead and message delivery delay by up to 82% and 45%, respectively, compared to SDNs using TCP for their southbound communication.

1. 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) (Jain et al., 2013), Network Function Virtualization (NFV) (Yousaf et al., 2017), wireless networks (Dezfouli et al., 2018; Kaloxylos, 2018), and edge and fog computing (Hu et al., 2015; Powell et al., 2020).

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 (RYU Controller, 2017) and OpenDayLight (ODL) (OpenDayLight Controller, 2021), provides functionalities such as network topology discovery, network operation analysis, and flow rule 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) (Case et al., 1990) to more advanced ones such as OpenFlow (McKeown et al., 2008), Open vSwitch Database (OVSDB) (Pfaff and Davie, 2013), and Network Configuration Protocol (NETCONF) (Enns et al., 2011).

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 (RYU, 2019a), Cisco-OpenFlow-Plugin (RYU, 2019b), HP OpenFlow (RYU, 2019c) are being used to configure flow rules and flow tables in switches via exchanging various 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 (Alsaeedi et al., 2019; Noormohammadpour and Raghavendra, 2017; Curtis et al., 2011b). This is primarily caused by the limited memory of switches and the arrival of new flows (Jouet et al., 2015). As the table miss rate increases, the overhead of the transport layer protocol used for communication between the controller and switches elevates (Ying et al., 2019; Kim et al., 2017). `Flow_mod` packets, which are used to install or modify flow rules, can be sent reactively

* Corresponding author.
E-mail addresses: pkumar@scu.edu (P. Kumar), bdezfouli@scu.edu (B. Dezfouli).

https://doi.org/10.1016/j.jnca.2023.103780
Received 9 July 2022; Received in revised form 4 July 2023; Accepted 24 October 2023
Available online 10 November 2023

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 (Chen et al., 2021). 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 (Palma et al., 2014; Caba and Soler, 2015; Sharma et al., 2014; Flathagen et al., 2018; Volpato et al., 2017; Chen et al., 2021).

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 enhance network visibility and 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 (Bosshart et al., 2014), APIs for accessing and configuring switches' data plane state (e.g., OpenConfig OpenConfig, 2016, NETCONF Enns et al., 2011), event management, Linux shell access, and the capability to run containers and Virtual Machines (VMs). For example, Arista's Extensible Operating System (EOS) (Arista, 2022) provides a Software Development Kit (SDK) for the 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-source protocols. 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.

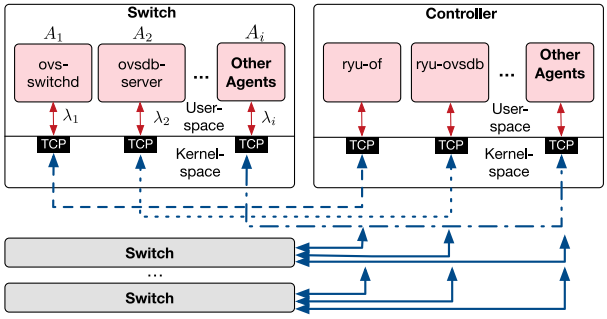
The wide range of programmability features available in today's switches reveals the increasing demand for communication between the control and data planes. This paper looks at southbound communication from a different perspective—*the transport layer protocols*. We argue that this communication imposes significant transport layer protocol overhead and delay that affect the utilization of bandwidth resources and network scalability (Karakus and Durresi, 2017; Hu et al., 2014). Since the exchange of messages between the control and data planes is crucial for network performance monitoring and configuration, a reliable transport layer protocol is required. In particular, factors such as packet congestion on switches may result in the loss of such messages. Additionally, as discussed in Isong et al. (2020), Das et al. (2019) and Xiao et al. (2014), controller and switches may reside in different networks and hops away, further increasing communication unreliability. Therefore, although User Datagram Protocol (UDP) is a low-overhead transport layer protocol, it cannot be used for southbound communication; instead, a transport layer protocol capable of providing reliability, resiliency towards out-of-order packets, and compatibility with security protocols such as Transport Layer Security (TLS), is required.

Currently, communication reliability, packet reordering, and security in SDNs are achieved by using Transmission Control Protocol (TCP) (combined with 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 on a switch are communicating with a controller, each agent needs to open its connection (due to the lack of supporting multiplexing connections by TCP), thereby increasing communication overhead (e.g., acknowledgment packets, connection establishment, and keep-alive packets) caused by the transport layer. Also, TCP does not deal with the Head-of-Line (HOL) blocking problem, which causes increased message delivery delay.

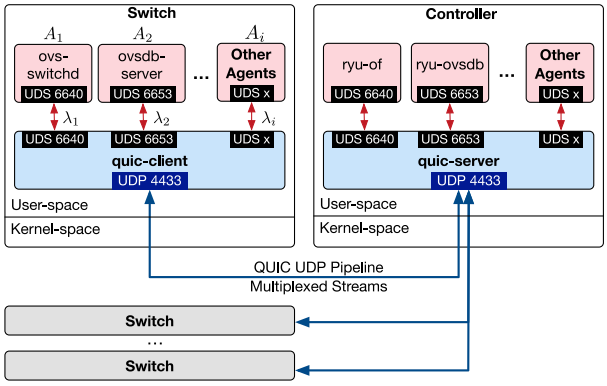
In this paper, we introduce *quicSDN*, a novel framework to address the drawbacks and challenges of using TCP as the transport layer protocol for southbound communication in SDNs. Specifically, instead of using TCP, quicSDN uses a new transport layer protocol called Quick UDP Internet Connection (QUIC) (Iyengar and Thompson, 2020). Although QUIC was primarily designed for web traffic (Hypertext Transfer Protocol-3 (HTTP3) HTT, 2019), 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 Acknowledgement (ACK) sent for original and retransmitted packets (a.k.a., TCP ambiguity problem). It is worth noting that since both QUIC and TCP use the widely-adopted and robust CUBIC congestion control algorithm (Iyengar and Swett, 2021; Rhee et al., 2018), we do not study the impact of congestion control algorithms on southbound communication. Instead, we differentiate and identify the essential shortcomings of TCP compared to QUIC for southbound communication. Towards proposing a novel architecture for SDNs, this paper presents the following contributions:

- We highlight the benefits of QUIC, compared to TCP, 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. This analysis shows that the overhead of QUIC drops compared to TCP when 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. The proposed framework also details the Inter-Process Communication (IPC) methods to allow multi-agent switches and controllers 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 available[1] and includes newly developed and modified entities of an SDN architecture. 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.

[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.

– 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 2 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 3. Section 4 discusses the implementation, algorithms, and pertinent details of quicSDN. Empirical evaluations are given in Section 5. Section 6 overviews the related work. We conclude the paper in Section 7.

2. 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.

2.1. 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.

Table 1
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

Fig. 1(a) 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 $\mathbf{A} = \{A_1, A_2, \ldots\}$ 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 $\mathbf{M}_{A_i} = \{m^i_1, m^i_2, \ldots\}$, where each element of the set represents the message size in bytes. Also, assume the intervals between message generations 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. As Fig. 1(a) shows, *each agent must be associated with its own socket*. Therefore, the minimum overhead of sending data from the sender to the receiver is:

$$\sum_{\forall A_i \in \mathbf{A}} ((H_{EP} + H_{IP} + H_{TCP}) \times \left\lceil \frac{\sum_{\forall m^i_j \in \mathbf{M}_{A_i}} m^i_j}{L_{MTU} - (H_{IP} + H_{TCP})} \right\rceil) \quad (1)$$

where m^i_j refers to the size of message j generated by agent A_i (in the application layer), 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 1.

In this paper, we propose and develop the *quicSDN* architecture, demonstrated in Fig. 1(b).[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 UDP pipeline. A stream is formatted as a *frame* inside a packet and represents a lightweight abstraction of server–client connection. Each stream 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:

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

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

[2] We will explain the implementation details in Sections 3 and 4.

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

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

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

Eqs. (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 (Mogul and Minshall, 2001), 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 ($\bar{m} = m_1^i = m_2^i = \cdots, \forall A_i \in \mathbf{A}$) 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 $\bar{B}_{A_i} = \{b_1, b_2, \ldots\}$ 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:

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

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 Fig. 1(b)) 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 $\hat{\mathbf{B}} = \{\hat{b}_1, \hat{b}_2, \ldots\}$, where each element $\hat{b}_j$ is the number of bytes in a burst. The overhead of QUIC is computed as follows:

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

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.

Fig. 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 1, 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 5.1.

2.2. 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 Fig. 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 5.2.

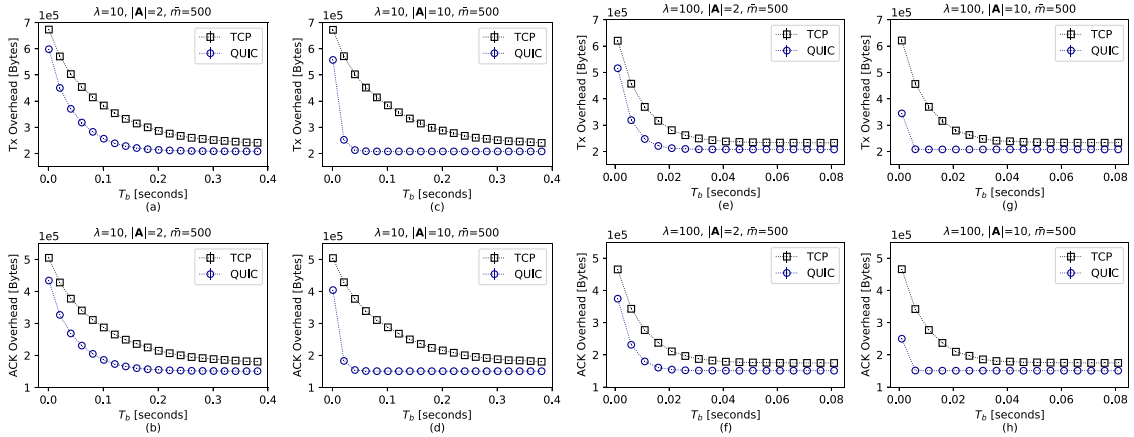


Fig. 2. Each column shows the transmission overhead and acknowledgment overhead of TCP and QUIC in various scenarios. The first row shows transmission overhead and the second row shows acknowledgment overhead. λ is rate of message generation by each agent. $|\mathbf{A}|$ is the number of agents. $\bar{m}$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

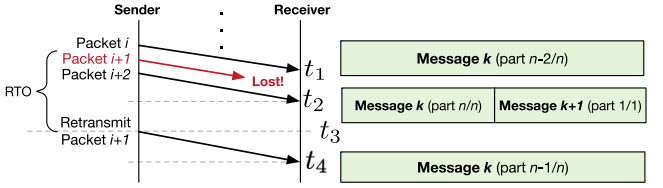


Fig. 3. The effect of HOL blocking on message delivery delay to application layer. In this scenario, the delivery of Message $k+1$ is at least $RTT + RTT/2$ slower when using TCP, compared to QUIC.

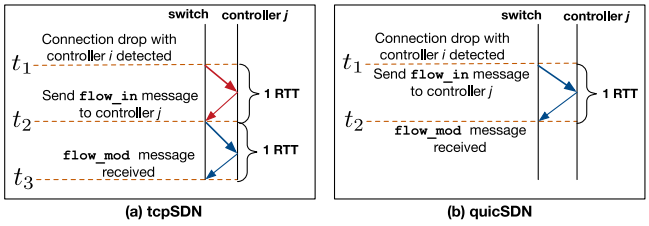


Fig. 4. Connection reestablishment delay between switches and controllers in (a) tcpSDN and (b) quicSDN. quicSDN facilities dynamic assignment of switches to controllers with a shorter delay and lower overhead.

2.3. Connection establishment and migration

Both TCP and QUIC need to establish a connection before data exchange. TCP with TLS1.2 (Dierks and Rescorla, 2008) and TLS1.3 (Rescorla, 2018) require three and two RTTs, respectively, for connection establishment. 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 is lower compared to tcpSDN. A sample scenario is given in Fig. 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×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 (Bera et al., 2020; Müller et al., 2014; Alowa and Fevens, 2019; Qin et al., 2018).

2.4. Congestion and flow control

QUIC's congestion control mechanism provides a richer set of features compared to TCP (Iyengar, 2016). 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 (Iyengar, 2020).

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.

3. 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 Fig. 1(b) 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.

3.1. 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.

3.1.1. Shared memory

To allow multiple applications (agents) to communicate through shared data structures, either they must be complied 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 (Odaira and Hiraki, 2003; Jung et al., 2014).

3.1.2. 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 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: 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.

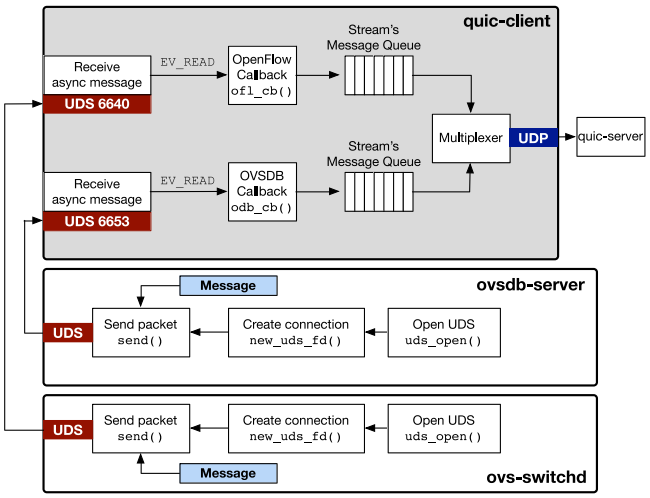


Fig. 5. The architecture of a switch in quicSDN. The figure highlights the modifications to OVS and how packets are processed by the quic-client component.

3.2. 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. Fig. 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.

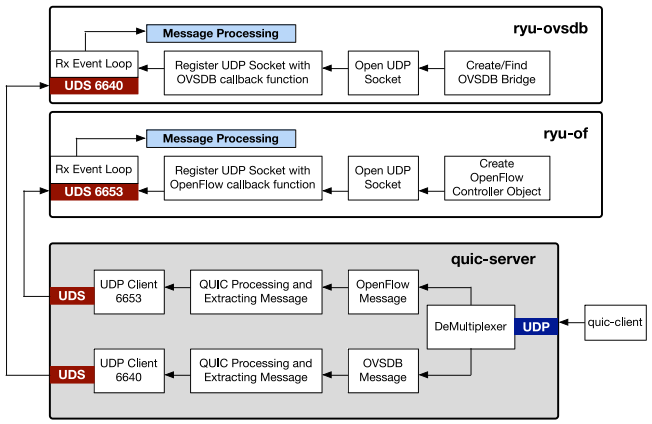


Fig. 6. The architecture of controller in quicSDN. This figure highlights the modifications to RYU and shows how packets are processed by the quic-server component.

The QUIC RFC (Iyengar and Thompson, 2020) 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.

3.3. Controller

Fig. 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 (George, 2021) 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.

4. 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 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.

4.1. 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 Section 3.2. 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.

4.2. 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 (Anon., 2021) 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.

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
           // 0-RTT Scenario
32         if !(resume()) then
33             return failure
34     else
           // 1-RTT Scenario
35         do_handshake()
36         if !(connect() then
37             return failure
       // Starting event loop
38     ev_run(ev_d, 0)
39     return
40 function readcb(ev_loop *loop, ev_io *w)
41     auto c = <client *>w→data;
42     on_read()
43 function writecb(ev_loop *loop, ev_io *w)
44     auto c = <client *>w→data
45     on_write()
46 function ofl_cb(ev_loop *loop, ev_io *w)
47     auto c = <ofl *>w→data
48     this→type_flag = openflow
49     on_ofl_odb_read()
50 function odb_cb(ev_loop *loop, ev_io *w)
51     auto c = <ofdb *>w→data
52     this→type_flag = ovsdb
53     on_ofl_odb_read();
```

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_recv(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 =
             generate_stream_id_divisisble_by_3()
40     else if (this→ovsdb) then
41         int stream_id = generate_normal_stream_id()
42     on_write_stream(stream_id, buf)
43     return
44 function on_write_stream(stream_id, buf)
45     while true do
46         auto n = _conn_write_stream(ndatalen)
47         if n > 0 && and ndatalen > 0 then
48             data.seek(ndatalen)
49         send_packet()
50         if buf.size() = 0 then
51             break
52     return
53 function on_ofl_odb_read()
54     array<uint8_t, 65536> buf_ofl
55     array<uint8_t, 65536> buf_odb
56     if activity detected of fp_ofl then
57         if (recvfrom(this→fp_ofl, buf_ofl.data, buf_ofl.len)) then
58             send_queue.push(buf_ofl)
59     if activity detected of fp_odb then
60         if recvfrom(this→fp_odb, buf_odb.data, buf_odb.len)) then
61             send_queue.push(buf_odb)
62             return
```

4.2.1. 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

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

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 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 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 (Schinazi and Rescorla, 2020) and 1-RTT keys negotiations, and the short header is used for subsequent data communications. `crypt_quic_message()` 4.2.3 (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).

4.2.2. 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>
```

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_ → readcd(), 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()
```

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 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 OVSDB 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).

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) for ;; do
23         n = _conn_write_pkt()
24         if n = 0 then
25             break
26         buf_push(n);
27         send_packet(buf)
28     return
29 function on_ofl_odb_read()
30     while true do
31         buffer<uint8_t, int, port> buf_ofl
32         if (recvfrom(this→fp_ofl, data, datalen)) then
33             buf_ofl.data = data
34             buf_ofl.len = datalen
35             buf_ofl.port = this→port;
               // e.g 6653 for openflow
36             send_queue.push(buf_ofl)
37         buffer<uint8_t, int, port> buf_odb;
38         if (recvfrom(this→fp_odb, data, datalen)) then
39             buf_odb.data = data
40             buf_odb.len = datalen
41             buf_odb.port = this→port
               // e.g 6640 for ovsdb
42             send_queue.push(buf_odb)
43     return
```

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

4.2.3. *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 (Rogaway, 2002) 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 $aux \in \{0, 1\}$. 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.

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
           sendto(this→fp_ofl)
17     else
18         _conn_map[odb_port] = stream_id sendto(this→fp_odb)
```

4.3. 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 <OVSDB 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.

5. Empirical evaluation

In this section, we empirically evaluate quicSDN versus tcpSDN. The testbed configuration is demonstrated in Fig. 7. We use three high-performance switches: one Arista 7050QX (Switch 3) and two Arista 7050CX3 (Switch 2 and 4). For Switch 1, we use a Linux machine,

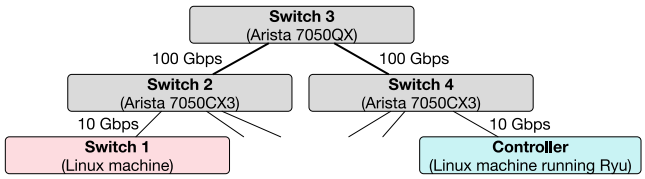


Fig. 7. Testbed topology. Switch 1 and the Controller communicate through Switch 2, 3, and 4.

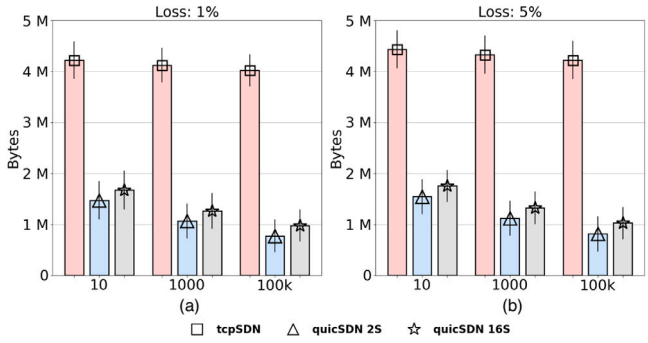


Fig. 8. The protocol overhead of tcpSDN and quicSDN in the *short RTT* scenario. Protocol overhead is the bidirectional communication overhead between controller and the switches. The *x*-axis shows three message generation rates: 10, 1000, 100 k messages per second. Sub-figures (a) and (b) present the protocol overhead for 1% and 5% packet loss rates, respectively. The notation 'M' stands for megabytes (2^{20}).

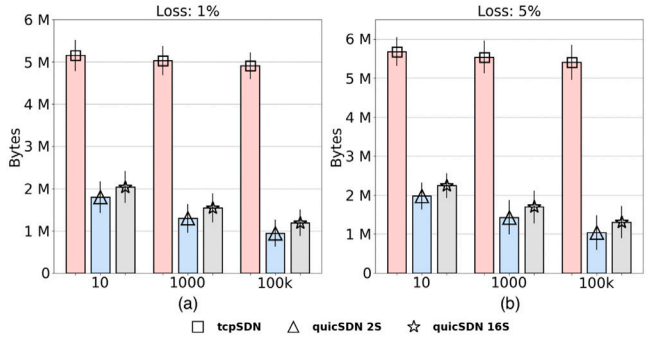


Fig. 9. The protocol overhead of tcpSDN and quicSDN in the *long RTT* scenario. Protocol overhead is the bidirectional communication overhead between controller and the switches. The *x*-axis shows three message generation rates: 10, 1000, 100 k messages per sec. Sub-figures (a) and (b) present the overhead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

which allows us to run OpenFlow and OVSDB and emulates the execution of more agents on the switch to generate various traffic patterns. 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. 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 3.2, each message is assigned a different stream identifier in the range 0 to 15 in a round-robin fashion.

5.1. 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 enforcement of the command. Therefore, we measure the *bidirectional* communication overhead between the two nodes. In the testbed, 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*. With long RTT, the testbed mimics a scenario where a remote controller controls the data plane or when considerable traffic competes with southbound control traffic. For each experiment, 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 second to 10, 1000, and 10,000. To represent various congestion levels in the switches, we introduce 1% and 5% packet loss rates. Figs. 8 and 9 present protocol overhead for the short-RTT and long-RTT scenarios, respectively. The overhead of quicSDN is up to 82% and 80% lower than tcpSDN in the short-RTT and long-RTT scenarios, respectively.

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, quicSDN shows a faster overhead decline than tcpSDN.

These empirical results conform to our analytical results in Section 2.1. 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 also increase, depending on the message size. Although a higher number of streams results in 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.

Figs. 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 is caused by the fluctuations in RTT, which result in out-of-order packets and more retransmissions. However, the overhead increase of quicSDN is lower than that of tcpSDN. For example, for 1% loss rate and ten messages/second, increasing the RTT caused the overhead of tcpSDN to increase by 25.9%, while the overhead of quicSDN-2s increased by 19.5%. TCP does not support NACK, instead it supports Selective Acknowledgement (SACK). While NACK specifies the packets that a receiver has not received, SACK specifies the range of packets a receiver has received. TCP allows including up to three SACK blocks (Mathis et al., 1996), whereas, QUIC supports 256 NACK blocks (Iyengar and Thompson, 2020). 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.

5.2. Message delivery delay

In this experiment, we measure message delivery delay between the Controller and Switch 1 in Fig. 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 the complete message reception by the application running on the receiver. We use two message size pairs: < 400,600>, and < 3000,600>. 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. Fig. 10 shows the result. The delivery delay of quicSDN is up to 27% and 45% better than tcpSDN for the 0 ms and 100 ms RTT scenarios, respectively.

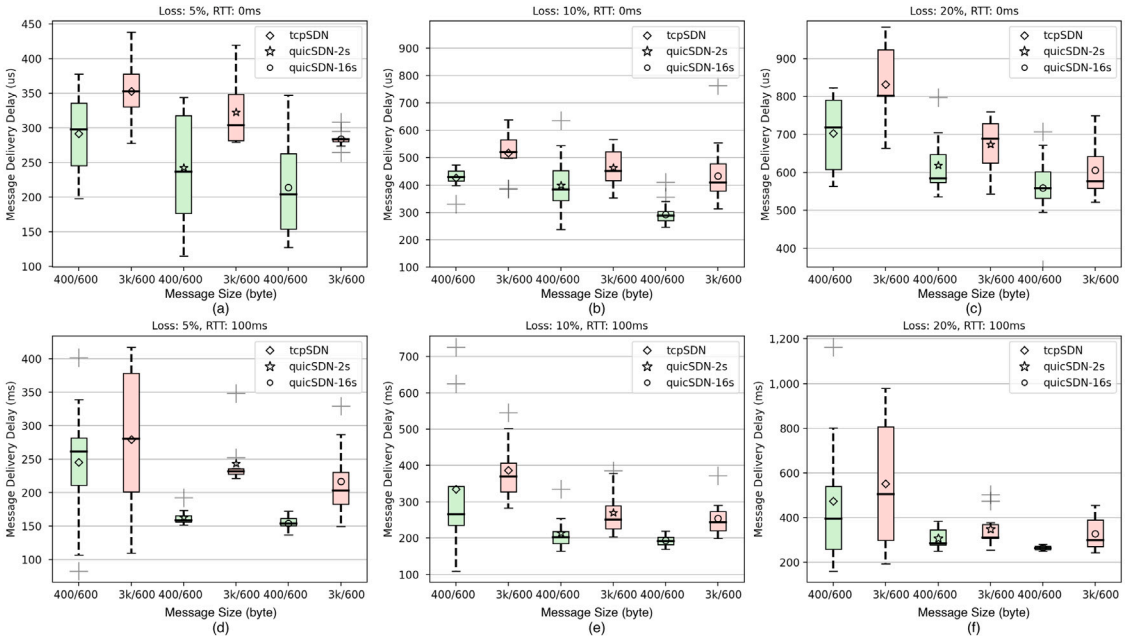


Fig. 10. Message delivery delay (ms). The experiment is performed 30 times for each configuration. The results confirm the lower message delivery delay of quicSDN compared to tcpSDN. Also, quicSDN-16s provides shorter delivery delay compared to quicSDN-2s because using a higher number of streams increases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

We observe that the message delivery delay of tcpSDN is higher than quicSDN-2s and quicSDN-16s. The primary reason behind this improvement 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 numbers. Also, the message delivery delay of quicSDN-2s is higher than quicSDN-16s. This is because, as the number of streams increases, QUIC can more effectively determine the independence between frames (inside packets). 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 < 400,600>. Fig. 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 (e.g., due to congestion on a switch) 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.[5] 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.

Fig. 11b presents a sample packet transmission from a sender to a receiver where the message sizes generated by the sender are < 3000,600>. 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 Fig. 10 that show delivery delay is higher for larger message sizes.

6. Related work

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

6.1. QUIC in SDNs

Works exist on using and enhancing QUIC flows for exchanging data and controlling traffic in SDNs. However, none of these works propose and implement a generic system architecture for southbound communication in SDNs, they do not present a mathematical analysis of the overhead of QUIC and TCP, and they mostly rely on emulation (through Mininet Contributors, 2022 or virtual machines) instead of real-world deployments. Table 2 highlights the major differences between quicSDN and these works.

Lau et al. (0000) proposed a message scheduling method for transmitting OpenFlow messages over QUIC. While some of the observations

[5] RTO value is usually higher than RTT to account for RTT variations.

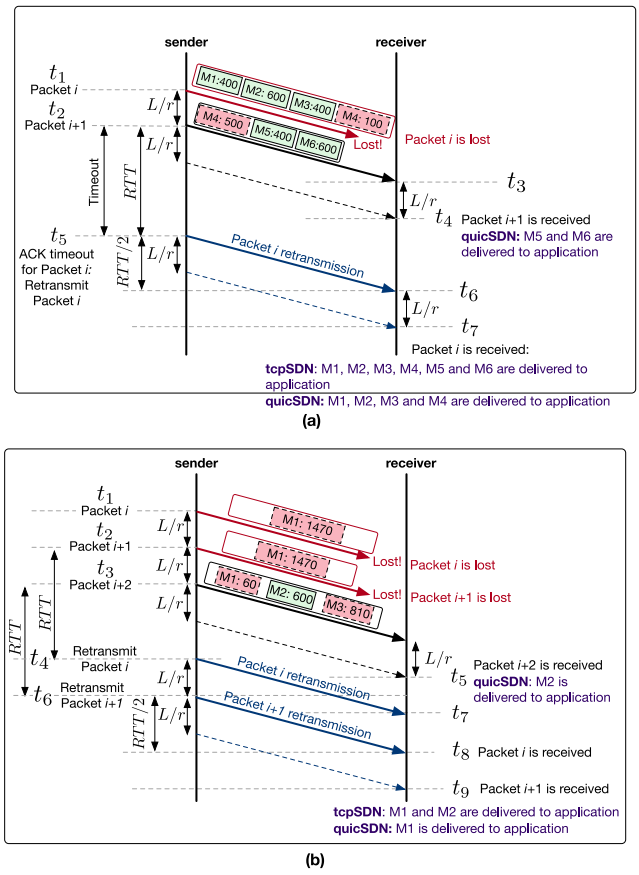


Fig. 11. The effect of packet loss and message size on the message delivery delay of tcpSDN and quicSDN. Here, L is the packet length (bits) and r is the transmission rate (bps) between the client and server. Red boxes show partial message inclusion in a packet and green boxes show complete inclusion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

in their work conform with ours, they did not present analytical modeling and analysis of the communication overhead and delay, as we did in Sections 2.1 and 5.2. Also, compared to Lau et al. (0000), the quicSDN architecture we proposed in this paper is a generic framework that supports multiplexing and can be used with various types of agents and protocols. Specifically, in Section 3, we identified system design challenges and proposed methods for architecting a generic framework. Rezende et al. (2019) introduced a SDN framework for handling multi-stream applications. For instance, whenever a user application needs to communicate with another machine through multiple Stream Control Transmission Protocol (SCTP) streams, the user application communicates with the controller first and requests to establish necessary rules supporting the required number of streams through switches. Considering the unique characteristics of QUIC data flows, Hussein et al. (2018) proposed a new controller component for handling the generation of flow rules for QUIC flows. For instance, the proposed controller component tries to assign QUIC flows to least-congested paths. Cohen et al. (2021) leverage the central network visibility of SDN controller to adjust network coding operation dynamically. They show that delivering video frames through the enhanced QUIC data flows (supporting dynamic coding) outperforms the regular QUIC protocol. Tong et al. (2021) proposed a method for classifying encrypted QUIC traffic and then using a SDN controller to assign these flows to paths that meet the Service Level Agreements (SLAs) requirements. Hayes et al. (2017) proposed an SDN-based method to dynamically manage the use of Multipath Transmission Control Protocol (MPTCP)'s paths and QUIC for media streaming to multi-homed client devices such as mobile phones. An application running on each client communicates with a SDN controller that provides the client with information about the RTT of communication paths. Based on the number and RTT of paths, the client dynamically selects a path between QUIC and MPTCP. Similarly, Bhat et al. (2018) leverage the multiplexing feature of QUIC for the transmission of high-quality video flows in SDNs. Guillen et al. (2019) proposed an SDN architecture to assist the end-points in rate adaptation using Dynamic Adaptive Streaming over HTTP (DASH) with QUIC and MPTCP. The main module in their controller architecture is the Network module, which performs traffic shaping and routing packets through the most efficient routes available. The SDN controller collects the priority of each user and their traffic, monitors end-to-end paths between peers, and performs users' traffic assignment to these paths based on their priority and required QoS levels.

6.2. 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 (Al-Fares et al., 2010) allows switches to handle mice flows via Equal-Cost Multi-Path (ECMP), and switches only consult the controller when dealing with elephant flows. DIFANE (Yu et al., 2010) distributes OpenFlow wildcards across switches to allow them to perform local routing. Curtis et al. (2011b) show that polling statistical data from switches reduces flow rule setup rate. 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 Addressable Memory (TCAM). Mahout (Curtis et al., 2011a) uses sender's TCP buffer size to identify mice flows and decide whether communication with the controller is necessary or not. Kim et al. (2014) 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. (2018) 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 Van Bemten et al. (2019) 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 increases, 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 (Koponen et al., 2010) 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 (Tootoonchian and Ganjali, 2010) synchronizes the status of distributed controllers and provides control applications with uniform, consistent access to the overall network data. Kandoo (Hassas Yeganeh and Ganjali, 2012) 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. (2020) propose

Table 2
Comparison of quicSDN and existing works on using QUIC in SDNs.

	Lau et al. (0000)	Rezende et al. (2019)	Hussein et al. (2018)	Cohen et al. (2021)	Tong et al. (2021)	Hayes et al. (2017)	Bhat et al. (2018)	Guillen et al. (2019)	**quicSDN**
QUIC for SDN southbound communication	√	X	X	X	X	X	X	X	√
Generic framework for multi-agent SDN southbound communication	X	X	X	X	X	X	X	X	√
Design and implementation of SDN southbound communication system	X	X	X	X	X	X	X	X	√
QUIC Transport for data plane flows	X	√	√	√	√	√	√	√	X
Load balancing of data plane flows	X	√	X	√	√	√	X	√	X
Mathematical overhead analysis of QUIC and TCP	X	X	X	X	X	X	X	X	√
Real-world testbed	X	X	X	X	X	X	X	X	√

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 (Phemius et al., 2014) 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. (2011) 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 (Müller et al., 2014) 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 (Zhang et al., 2011). Beheshti and Zhang (2012) 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.

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.

7. Conclusion

As the need for a higher rate, lower overhead, and low-latency 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 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 (AI-powered) configurations to enhance network visibility, performance, and security. Third, considering the specifics of southbound control flows, a study of the effects of various congestion control algorithms on the delay and throughput of southbound communication flows is a direction for future work. Fourth, 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 (which perform packet switching between containers, VMs and external devices), the proposed switch architecture may be integrated with Data Plane Development Kit (DPDK) to reduce inter-process communication overhead. Fifth, throughput and latency can be further improved by bringing Ethernet, Internet Protocol (IP), and UDP packet processing into userspace for QUIC packets. Finally, quicSDN is a generic architecture that can be used in other systems, such as the communication of Internet of Things (IoT) devices with gateways or controllers with gateways. In such systems, the proposed framework can be adopted to enhance energy efficiency, in addition to lowering communication overhead and delay.

CRediT authorship contribution statement

Puneet Kumar: Conceptualization, Investigation, Software, Visualization, Validation, Data curation, Methodology, Writing – original draft, Writing – reviewing & editing. **Behnam Dezfouli:** Conceptualization, Methodology, Formal analysis, Investigation, Supervision, Writing – original draft, Writing – reviewing & editing, Resources, Funding acquisition.

Declaration of competing interest

The authors have no conflict of interest to disclose.

Data availability

Data will be made available on request.

Acknowledgment

This work has been partially supported by a gift fund and equipment donation from Arista Networks (Santa Clara, United States).

References

Bishpop, M. (Ed.), 2019. Hypertext transfer protocol version 3 (HTTP/3). URL https://tools.ietf.org/html/draft-ietf-quic-http-32.

Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., Vahdat, A., et al., 2010. Hedera: Dynamic flow scheduling for data center networks. In: NSDI, vol. 10, (8), pp. 89–92.

Alowa, A., Fevens, T., 2019. 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, pp. 269–276.

Alsaeedi, M., Mohamad, M.M., Al-Roubaiey, A.A., 2019. Toward adaptive and scalable OpenFlow-SDN flow control: A survey. IEEE Access 7, 107346–107379.

Anon., 2021. ngtcp2 project. URL https://github.com/ngtcp2/ngtcp2.

Arista, 2022. Arista extensible operating system (EOS), the world's most advanced network operating system. URL https://www.arista.com/en/products/eos.

Beheshti, N., Zhang, Y., 2012. Fast failover for control traffic in software-defined networks. In: IEEE Global Communications Conference, GLOBECOM. IEEE, pp. 2665–2670.

Bera, S., Misra, S., Saha, N., 2020. Traffic-aware dynamic controller assignment in SDN. IEEE Trans. Commun. 68 (7), 4375–4382.

Bhat, D., Deshmukh, R., Zink, M., 2018. Improving qoe of abr streaming sessions through quic retransmissions. In: Proceedings of the 26th ACM International Conference on Multimedia. pp. 1616–1624.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al., 2014. P4: Programming protocol-independent packet processors. ACM SIGCOMM Comput. Commun. Rev. 44 (3), 87–95.

Caba, C., Soler, J., 2015. APIs for QoS configuration in software defined networks. In: Proceedings of the 1st IEEE Conference on Network Softwarization, NetSoft. pp. 1–5.

Case, J.D., Fedor, M., Schoffstall, M.L., Davin, J., 1990. RFC1157: Simple network management protocol (SNMP).

Chen, J., Gopal, A., Dezfouli, B., 2021. Modeling control traffic in software-defined networks. In: 2021 IEEE 7th International Conference on Network Softwarization, NetSoft. IEEE, pp. 258–262.

Cohen, A., Esfahanizadeh, H., Sousa, B., Vilela, J.P., Luis, M., Raposo, D., Michel, F., Sargento, S., Medard, M., 2021. Bringing network coding into SDN: Architectural study for meshed heterogeneous communications. IEEE Commun. Mag. 59 (4), 37–43.

Contributors, M.P., 2022. Mininet an instant virtual network on your laptop (or other PC). URL https://mininet.org/.

Curtis, A.R., Kim, W., Yalagandula, P., 2011a. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In: IEEE Conference on Computer Communications, INFOCOM. pp. 1629–1637.

Curtis, A.R., Mogul, J.C., Tourrilhes, J., Yalagandula, P., Sharma, P., Banerjee, S., 2011b. DevoFlow: Scaling flow management for high-performance networks. In: ACM SIGCOMM. pp. 254–265.

Das, T., Sridharan, V., Gurusamy, M., 2019. A survey on controller placement in SDN. IEEE Commun. Surv. Tutor. 22 (1), 472–503.

Dezfouli, B., Esmaeelzadeh, V., Sheth, J., Radi, M., 2018. A review of software-defined WLANs: Architectures and central control mechanisms. IEEE Commun. Surv. Tutor. 21 (1), 431–463.

Dierks, T., Rescorla, E., 2008. The transport layer security (TLS) protocol version 1.2.

Enns, R., Bjorklund, M., Schoenwaelder, J., Bierman, A., 2011. Network configuration protocol (NETCONF).

Flathagen, J., Mjelde, T.M., Bentstuen, O.I., 2018. 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. pp. 1–6.

George, E., 2021. Eventlet open source project profile. URL https://github.com/eventlet/eventlet/blob/master/eventlet/greenio/base.py.

Guillen, L., Izumi, S., Abe, T., Suganuma, T., 2019. Sand/3: SDN-assisted novel qoe control method for dynamic adaptive streaming over http/3. Electronics 8 (8), 864.

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

Hayes, B., Chang, Y., Riley, G., 2017. Omnidirectional adaptive bitrate media delivery using mptcp/quic over an sdn architecture. In: GLOBECOM 2017-2017 IEEE Global Communications Conference. IEEE, pp. 1–6.

Hu, J., Lin, C., Li, X., Huang, J., 2014. Scalability of control planes for software defined networks: Modeling and evaluation. In: IEEE 22nd International Symposium of Quality of Service, IWQoS. pp. 147–152.

Hu, Y.C., Patel, M., Sabella, D., Sprecher, N., Young, V., 2015. Mobile edge computing—A key technology towards 5G. ETSI White Pap. 11 (11), 1–16.

Hussein, A., Kayssi, A., Elhajj, I.H., Chehab, A., 2018. SDN for QUIC: An enhanced architecture with improved connection establishment. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing. pp. 2136–2139.

Isong, B., Molose, R.R.S., Abu-Mahfouz, A.M., Dladlu, N., 2020. Comprehensive review of SDN controller placement strategies. IEEE Access 8, 170070–170092.

Iyengar, J., 2016. QUIC at 10,000 feet. URL https://docs.google.com/document/d/1gY9-YNDNAB1eip-RTPbqphgySwSNSDHLq9D5Bty4FSU/edit.

Iyengar, J., 2020. QUIC at 10,000 feet. URL https://tools.ietf.org/html/draft-ietf-quic-recovery-33.

Iyengar, J., Swett, I., 2021. RFC 9002: QUIC loss detection and congestion control.

Iyengar, J., Thompson, M., 2020. QUIC: A UDP-based multiplexed and secure transport. URL https://datatracker.ietf.org/doc/html/rfc9000.

Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., et al., 2013. B4: Experience with a globally-deployed software defined WAN. ACM SIGCOMM Comput. Commun. Rev. 43 (4), 3–14.

Jouet, S., Cziva, R., Pezaros, D.P., 2015. Arbitrary packet matching in OpenFlow. In: 2015 IEEE 16th International Conference on High Performance Switching and Routing, HPSR. IEEE, pp. 1–6.

Jung, H., Han, H., Fekete, A., Heiser, G., Yeom, H.Y., 2014. A scalable lock manager for multicores. ACM Trans. Database Syst. 39 (4), 1–29.

Kaloxylos, A., 2018. A survey and an analysis of network slicing in 5G networks. IEEE Commun. Stand. Mag. 2 (1), 60–65.

Karakus, M., Durresi, A., 2017. A survey: Control plane scalability issues and approaches in software-defined networking (SDN). Comput. Netw. 112, 279–293.

Kim, E.-D., Choi, Y., Lee, S.-I., Kim, H.J., 2017. Enhanced flow table management scheme with an LRU-based caching algorithm for SDN. IEEE Access 5, 25555–25564.

Kim, E.-D., Lee, S.-I., Choi, Y., Shin, M.-K., Kim, H.-J., 2014. A flow entry management scheme for reducing controller overhead. In: 16th International Conference on Advanced Communication Technology. IEEE.

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

Lau, W., Wong, K., Cui, L., 0000. Optimizing the performance of openflow protocol over quic, Available at SSRN 4348216.

Mathis, M., Mahdavi, J., Floyd, S., Romanow, A., 1996. RFC2018: TCP selective acknowledgement options.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J., 2008. OpenFlow: Enabling innovation in campus networks. ACM SIGCOMM Comput. Commun. Rev. 38 (2), 69–74.

Mogul, J.C., Minshall, G., 2001. Rethinking the TCP nagle algorithm. ACM SIGCOMM Comput. Commun. Rev. 31 (1), 6–20.

Müller, L.F., Oliveira, R.R., Luizelli, M.C., Gaspary, L.P., Barcellos, M.P., 2014. Survivor: An enhanced controller placement strategy for improving SDN survivability. In: IEEE Global Communications Conference. pp. 1909–1915.

Noormohammadpour, M., Raghavendra, C.S., 2017. Datacenter traffic control: Understanding techniques and tradeoffs. IEEE Commun. Surv. Tutor. 20 (2), 1492–1525.

Odaira, R., Hiraki, K., 2003. Selective optimization of locks by runtime statistics and just-in-time compilation. In: Proceedings International Parallel and Distributed Processing Symposium. IEEE, p. 6.

OpenConfig, 2016. OpenConfig: Vendor-neutral, model-driven network management designed by users. URL https://www.openconfig.net.

OpenDayLight Controller, 2021. URL https://www.opendaylight.org/.

Palma, D., Goncalves, J., Sousa, B., Cordeiro, L., Simoes, P., Sharma, S., Staessens, D., 2014. The queuepusher: Enabling queue management in openflow. In: Third European Workshop on Software Defined Networks. IEEE, pp. 125–126.

Pfaff, B., Davie, B., 2013. The open vswitch database management protocol. Internet Requests for Comments, RFC Editor, RFC 7047.

Phemius, K., Bouet, M., Leguay, J., 2014. Disco: Distributed multi-domain sdn controllers. In: IEEE Network Operations and Management Symposium, NOMS. pp. 1–4.

Powell, C., Desiniotis, C., Dezfouli, B., 2020. The fog development kit: A platform for the development and management of fog systems. IEEE Internet Things J. 7 (4), 3198–3213.

Qin, Q., Poularakis, K., Iosifidis, G., Tassiulas, L., 2018. SDN controller placement at the edge: Optimizing delay and overheads. IEEE Conf. Comput. Commun. 684–692.

Rescorla, E., 2018. The transport layer security (TLS) protocol version 1.3. URL https://tools.ietf.org/html/rfc8446.

Rezende, P., Kianpisheh, S., Glitho, R., Madeira, E., 2019. An SDN-based framework for routing multi-streams transport traffic over multipath networks. In: ICC 2019-2019 IEEE International Conference on Communications, ICC. IEEE, pp. 1–6.

Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., Scheffenegger, R., 2018. CUBIC for Fast Long-Distance Networks. Tech. Rep..

Rogaway, P., 2002. Authenticated-encryption with associated-data. In: Proceedings of the 9th ACM Conference on Computer and Communications Security. pp. 98–107.

RYU, 2019a. Arista directflow SDK. URL http://aristanetworks.github.io/EosSdk/docs/1.7.0/ref/directflow.html.

RYU, 2019b. Cisco OpenFlow plugin. URL https://www.cisco.com/c/en/us/td/docs/switches/datacenter/sdn/configuration/b_openflow_agent_nxos/b_openflow_agent_nxos_chapter_01.html.

RYU, 2019c. HP OpenFlow. URL https://support.hpe.com/hpesc/public/docDisplay?docLocale=en_US&docId=emr_na-c04777809.

RYU Controller, 2017. URL https://ryu.readthedocs.io/en/latest/getting_started.html.

Schinazi, D., Rescorla, E., 2020. QUIC version negotiation. URL https://tools.ietf.org/html/draft-ietf-quic-version-negotiation-02.

Sharma, S., Staessens, D., Colle, D., Palma, D., Goncalves, J., Figueiredo, R., Morris, D., Pickavet, M., Demeester, P., 2014. Implementing quality of service for the software defined networking enabled future internet. In: Third European Workshop on Software Defined Networks. IEEE, pp. 49–54.

Tong, V., Souihi, S., Tran, H.A., Mellouk, A., 2021. SDN-based application-aware segment routing for large-scale network. IEEE Syst. J. 16 (3), 4401–4410.

Tootoonchian, A., Ganjali, Y., 2010. Hyperflow: A distributed control plane for openflow. In: Proceedings of the Internet Network Management Conference on Research on Enterprise Networking, vol. 3. USENIX.

Van Bemten, A., Đerić, N., Varasteh, A., Blenk, A., Schmid, S., Kellerer, W., 2019. Empirical predictability study of SDN switches. In: ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS. pp. 1–13.

Volpato, F., Da Silva, M.P., Gonçalves, A.L., Dantas, M.A.R., 2017. An autonomic QoS management architecture for software-defined networking environments. In: IEEE Symposium on Computers and Communications, ISCC. pp. 418–423.

Xiao, P., Qu, W., Qi, H., Li, Z., Xu, Y., 2014. The SDN controller placement problem for WAN. In: 2014 IEEE/CIC International Conference on Communications in China, ICCC. IEEE, pp. 220–224.

Ying, R., Jia, W.-K., Luo, C., Wu, Y., 2019. Expedited eviction of invalid flow entries for SDN-based EPC networks. In: IEEE/CIC International Conference on Communications in China, ICCC. pp. 298–303.

Yousaf, F.Z., Bredel, M., Schaller, S., Schneider, F., 2017. NFV and SDN—Key technology enablers for 5G networks. IEEE J. Sel. Areas Commun. 35 (11), 2468–2478.

Yu, M., Rexford, J., Freedman, M.J., Wang, J., 2010. Scalable flow-based networking with DIFANE. ACM SIGCOMM 40 (4), 351–362.

Zhang, Y., Beheshti, N., Tatipamula, M., 2011. On resilience of split-architecture networks. In: IEEE Global Telecommunications Conference, GLOBECOM. pp. 1–6.

Puneet Kumar is currently pursuing his Ph.D. in the Department of Computer Science and Engineering, Santa Clara University. Puneet Kumar received his master's degree in Telecommunication Engineering from University of Maryland College Park, US. He has more than ten years of industrial experience during which he has worked for Cisco Systems, Calix Inc., Symantec Corp., Broadcom Inc., and Apple Inc. Puneet Kumar's research interests are transport layer protocols.

Behnam Dezfouli is an Associate Professor in the Department of Computer Science and Engineering and serves as the Director of SCU IoT Research Lab (SIOTLAB) at Santa Clara University (SCU). Prior to his role at SCU, he held positions as a Postdoctoral Research Scientist and Visiting Assistant Professor at the University of Iowa in the United States from 2015 to 2016. In 2014, Behnam Dezfouli was a Postdoctoral Research Fellow at University Technology Malaysia, Malaysia, and in 2012, he was a Research Fellow at the Institute for Infocomm Research in Singapore. He earned his Ph.D. in Computer Science from the University Technology Malaysia in 2014. He has recently contributed to projects at Infineon Technologies and HPE labs. His research interests encompass Internet of Things, software-defined networking and virtualization, edge and fog computing, and cyber–physical systems.