IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 17, 2017, accepted April 9, 2017, date of publication April 12, 2017, date of current version May 17, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2693376

SDUDP: A Reliable UDP-Based Transmission Protocol Over SDN

MING-HUNG WANG[1], (Student Member, IEEE), LUNG-WEN CHEN[2], PO-WEN CHI[3], (Member, IEEE), AND CHIN-LAUNG LEI[1]

[1]Department of Electrical Engineering, National Taiwan University, Taipei 10617, Taiwan
[2]MediaTek Inc., Hsinchu 30078, Taiwan
[3]Arcadyan Technology Corporation, Hsinchu 30071, Taiwan

Corresponding author: Ming-Hung Wang (d00921027@ntu.edu.tw)

This work was supported by the Ministry of Science and Technology, Taiwan, under Grant MOST 104-2221-E-002-099-MY3.

ABSTRACT The recent rapid development of Web technology, multimedia content, and interactive data has considerably expanded the size of the Internet transmissions. Benefiting from the paradigm-shifting technology of software defined networking (SDN), the administrators are now able to easily manage network flows by customizing flow rules over SDN. Inspired by this, we propose a UDP-based reliable transmission framework to improve efficiency of transmission control protocol (TCP) transmission on an SDN-enabled network. The main idea of our framework is to convert the TCP transmission into UDP packets to decrease the overhead during communications, such as handshaking, acknowledgment, and header overhead while using TCP. To guarantee reliability, we have leveraged the power of SDN to designate packets under our protocol to flow in predefined routes and monitor them to avoid possible packet loss. Our proposal is composed of a series of designs and implementations, including the packet format transformations, packet buffering, and retransmission mechanisms on switches. For users, this means that they are transmitting data with TCP, while the overhead of the TCP traffic is reduced significantly through a reliable and lightweight UDP transmission mechanism on the SDN-enabled network. Our evaluation results show that our framework provides a more efficient bandwidth usage and guarantees the reliability of packets as in TCP transmissions.

INDEX TERMS TCP improvement, reliable UDP, SDN, OpenFlow.

I. INTRODUCTION

In recent years, network bandwidth has become a crucial issue for both Internet service providers as well as content providers. For example, the increasing amount of multimedia content has occupied a large portion of available network capacity. However, most of the content is transmitted using TCP [1] as its genuine features, such as the slow start, the bandwidth probing, and the congestion control provide reliability during communication. Meanwhile, the overhead of TCP packet headers and ACK messages remains a major issue as it consumes more bandwidth and time than UDP transmissions [2].

Recently, Software Defined Networking (SDN) [3]–[8] has emerged as a more efficient new technology for managing both wired and wireless networks, as it decouples data and control planes and provides a centralized management infrastructure. SDN enables administrators to conduct lightweight monitoring and management of large-scale networks. For instance, SDN can provide offloading the traffic on specific routes through setting a series of flow routes for a certain service. In addition, the centralized management infrastructure can help network administrators with monitoring the status of a network and enable them to conduct instant actions when issues appear.

Motivated by the capabilities of SDN, this paper leverages the strength of SDN and UDP and proposes an enhanced transmission infrastructure. In our design, neither host has to be fundamentally changed. Rather, we implement our design on the SDN controller and the SDN switches to provide a reliable and efficient transmission. Through experiments, we have evaluated the effectiveness of our framework compared to TCP transmissions at many aspects such as the average load of switches in terms of bytes and number of packets. The results demonstrate that our framework outperforms TCP in different network environments and settings.

To sum up, our research achieves the following goals:

1) **Decrease transmission overhead.** We modify the transmission of TCP transmission by converting TCP packets into UDP packets among the SDN.

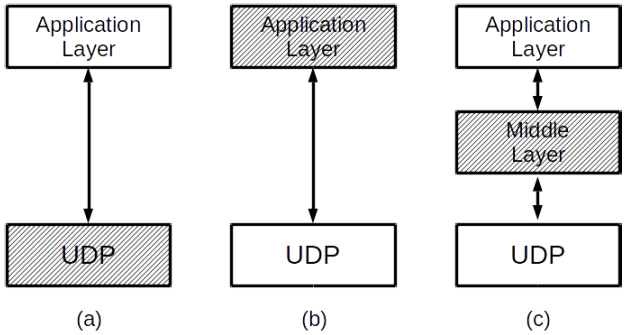

FIGURE 1. Three types of reliable UDP schemes. The gray blocks are used to indicate the modification parts of different scheme types.

Headers are simplified and acknowledgment packets are significantly reduced.

2) **Guarantee transmission reliability.** To retain TCP transmission reliability, our framework leverages the SDN function to monitor the packet flow of the network and retransmit packets while detecting packet loss.
3) **Maximize data frame utilization.** We propose a mechanism of packet accumulation to maximize the utilization of each data frame sent to the network.

This paper is organized as follows. Section 2 reviews the related works about reliable transmission using non-TCP mechanisms. Section 3 introduces the proposed SDUDP framework and includes details of components and designs. Section 4 presents our evaluation results and analyses, and in addition provides system performances and compares and contrasts them from different angles. Finally, Section 5 draws the conclusions and suggests future work on this subject.

II. RELATED WORKS

A. RELIABLE UDP

Reliable UDP is a long-investigated topic in networks. Compared to TCP, UDP is a lightweight transport protocol and is suitable for fast and efficient transmission without guarantees. However, sometimes users may encounter slight overhead for some additional features, like reliability. To leverage the strength of both protocols, many reliable UDP works have been proposed. These works can be divided into three types; transport layer modification, application layer modification, and middle layer modification (see Figure 1).

The first type of reliable UDP modifies the transport layer. That is, this approach changes the UDP's header and its behavior. In 1999 Cisco proposed the representative RELIABLE UDP PROTOCOL (RUDP) [9], which adds six additional bytes on a header for the acknowledgment, the retransmission, and the checksum. Although these added mechanisms are similar to TCP, the overhead is decreased. Note that RUDP is not currently a formal standard. Another example is DTLS [10]–[12]. DTLS is a secure protocol for datagram transmission instead of the transport layer. However, it provides a suitable example to see how UDP could deal with the reordering and packet loss issues. The advantage of this kind of approach is that the new UDP protocol can serve existing applications with more features. The trade-off is that users can only communicate with those who have the same transport layer protocol.

The second approach moves the works for reliability from the transport layer to the application layer. In 2002, He et al. proposed RBUDP (Reliable Blast UDP) [13] for QoS-enabled networks. They used TCP and UDP simultaneously with TCP used for signaling and UDP for data traffic. The TCP connection exchanged the data transfer status for ensuring reliability and QoS while the UDP channel, which has higher bandwidth, made data transfer more efficient. SABUL [14], which was proposed in 2003, is another similar approach for congestion control. Finally, Tsunami [15] is an approach which replaces sliding window by inter-packet delay adjustment for rate control. The advantage of these schemes is that the network stack remains unchanged and that therefore these approaches can be ported to almost all systems. The problem is that existing applications need to be patched and re-implemented.

The third approach builds a middle layer between the transport layer and the application layer. In 2007, Gu and Grossman proposed the UDT (UDP-based data transfer) [16] framework. This framework creates a UDT socket layer by providing applications with a new set of socket APIs while internally using OSI socket APIs. The reliability and congestion control mechanisms are implemented in this new socket layer. RUFC [17] is another framework that similarly creates a middle layer. What makes it different is that its middle layer can accept different policies for different applications. Thus, RUFC makes it possible to customize policies for application optimization. The obvious benefit of this design is that it can be applied directly to existing UDP. However, applications need to use another set of socket APIs. Another problem is that this type of approach adds a new layer which will cost extra overhead for data bypass.

Unfortunately, all of the above schemes are end-to-end solutions, meaning that in order to implement them, existing protocols need to be modified, no matter whether the modified part is in the transport layer, the application layer or between these two layers. In the real world that the TCP/IP has been well deployed, making it almost impossible to replace the transport layer of all hosts. Therefore, normal users cannot enjoy the benefits of reliable UDP. To fill in this gap this work imitates TCP behavior in UDP with SDN support so that users can setup a TCP connection while actually it is a UDP transmission in the network. This work is totally network-realized and is transparent to users.[1]

B. SPLIT TCP

The split TCP [18], [19] constructs TCP proxies in the middle of transmission to enable packet buffering upon receipt

[1]Transparency here does not mean that users do not know they are using this service but means that they do not modify the network protocol stack on their computers.

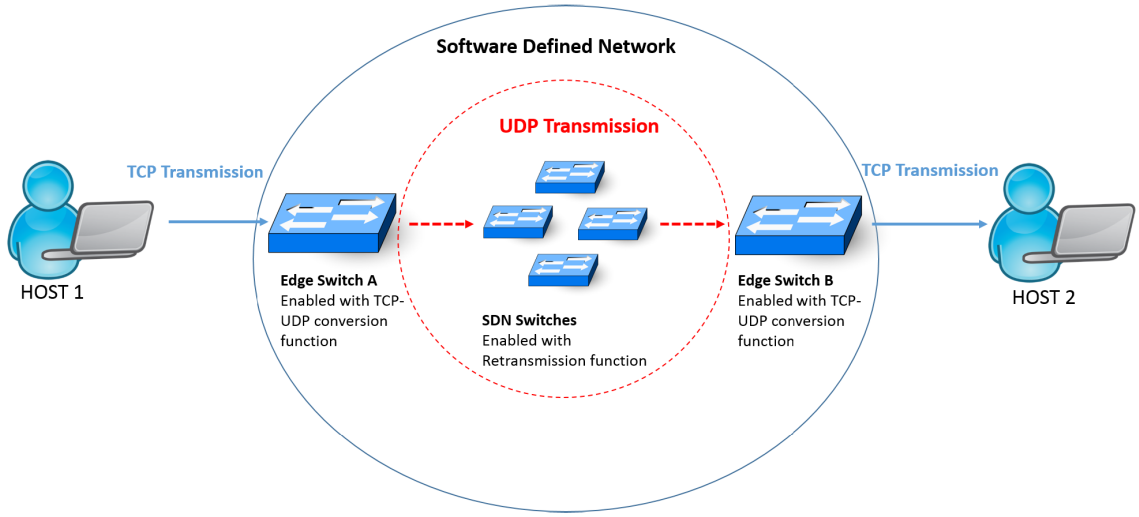

FIGURE 2. The proposed SDUDP transmission framework.

and transmission rate control. Benefiting from the buffering, dropped packets can be recovered from the most recent node. In addition, the rate control feature addresses the potential congestion through controlling rates of inter-proxy segments. Recent scholars [20], [21] have focused on improving the split-TCP concept and applied it to wireless network environments. SDUDP similarly leverages the concept of adding Retransmission Engines during the path. However, the major difference between SDUDP and the previous works of split TCP is that SDUDP leverages the power of SDN to monitor the IP Identification field for packet loss detection. Once SDUDP finds packet drop events according to the discontinuous value in Identification fields, the retransmission engine will start retransmission as soon as possible; thus, there is no need to wait for a timeout in order to start retransmission.

C. RELIABLE COMMUNICATION IN SDN

Although SDN has been adopted in improving network communication and management by previous scholars [22]–[25], not so many researches exist about achieving traffic reliability. The reason is that SDN is commonly treated as an infrastructure and reliability is usually guaranteed through end-to-end approaches, like discussed in the previous subsection. However, in multicast scenarios, SDN does play an important role in maintaining reliability. To deal with this issue previous scholars proposed the File Multicast Transport Protocol (FMTP) in [26]–[28], which uses in-sequence delivery on VLAN [29] to detect packet loss.

FMTP has one serious limitation, though, which is that its deployment requires VLAN support. Since VLAN is a layer-2 technology, it is impossible to apply it to communications over multiple networks.[2] To address this problem, our work makes use of IP information instead of VLAN information, which renders our approach usable across multiple networks. Besides, FMTP needs to be deployed both on senders and receivers instead of only on the SDN controller. That is, all drawbacks discussed in the previous subsection also stand here. As stated before, our method requires nothing changed on each host. To provide reliability we only implement our framework on the SDN controller and the SDN switches.

[2]Undoubtedly, by VPN technologies, multiple networks can be integrated as one local area network and VLAN can be used in this case.

III. PROPOSED FRAMEWORK

A. OVERVIEW

Figure 2 shows our proposed framework. In the figure, each switch represents an enhanced SDN switch, meaning that it possesses SDN switch capabilities, and our proposed engines. Edge Switches consist of the TCP Engine, the Retransmission Engine, and the Packet Sending Engine; while other SDN switches are enabled with the Retransmission Engines. Figure 3 shows the design of the Edge Switch.

As in Figure 3, the Edge Switch, the TCP Engine, and the Retransmission Engine take over two tasks: 1) transform packets from TCP to UDP (on sender side) and UDP to TCP (on receiver side) for transmission, and; 2) guarantee the reliability of transmission without using sequence numbers and ACK messages.

The TCP Engine is responsible for maintaining information of TCP connections between hosts, and furthermore, for transforming packets from the TCP format to the UDP format and from the UDP to the TCP format. The Retransmission Engine is designed for detecting and dealing with packet loss events through the Identification [30], [31] field of packets in the IP layer. To ensure that the packets sent to/from the TCP Engine can pass through the switches with the Retransmission Engine, we have deployed pre-defined flow rules [32], [33] in these switches to fix the path of these packets. Through the cooperation between these engines and the pre-defined flows, our framework guarantees sequentiality and reliability of data. Consequently, data can be transmitted

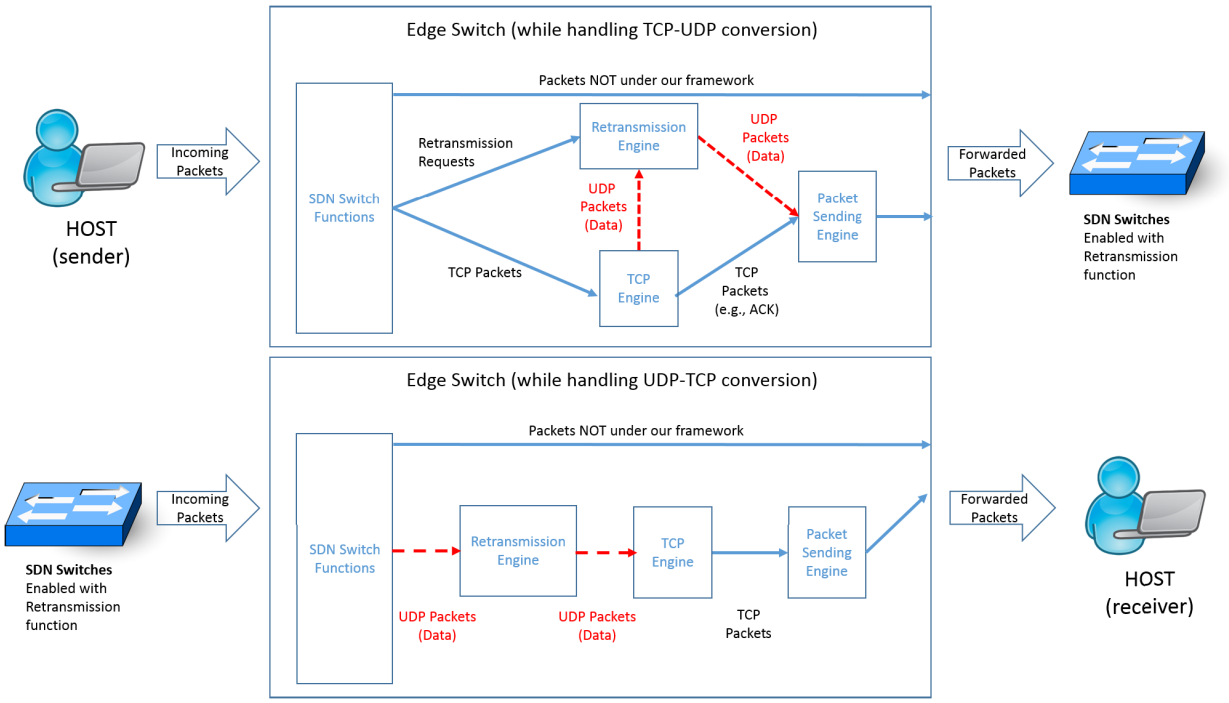

FIGURE 3. Scenarios in Edge Switches (sender side and receiver side).

through UDP over the network, which implies that our framework transmits packets with shorter headers and no ACK messages.

The following describes a sample transmission scenario of our framework. Given two hosts, Host 1 and Host 2. Host 1 likes to transmit data via a TCP connection to Host 2. Between Host 1 and Host 2, there are two nearest switches, we call these the "Edge Switches" and denoted them as Edge Switch A and Edge Switch B. Before the transmission starts, the initiation of the connection should set flow rules to establish a pre-defined route between the two switches to guarantee a fixed route for the connection.

In the fixed route, the Edge Switches will first deliver all the TCP packets that belong to the connection to the TCP Engine implemented in Edge Switch A. Then the TCP engine will decapsulate the TCP packet, retrieve the data, encapsulate them as UDP packets, and send them to Host 2. According to the pre-defined flows, the packets between Edge Switch A and Edge Switch B are then transmitted on a fixed path. Through this fixed path we assure that packets are transmitted orderly. Even though some packets may be lost during transmission, in our design the switches with the Retransmission Engine on the fixed path will detect and retransmit the lost packets.

The following six steps describe the establishment of communication:

1) **SYN message sent by Host 1.** Once TCP Engine in Edge Switch A (the one we call TCP Engine A) identifies a SYN message (from Host 1) that Host 1 wants to establish a TCP connection with Host 2, TCP Engine A starts the following procedure.
2) **Exchange information of Host 1 and 2.** Once TCP Engine A retrieves the information of Host 1 from SYN messages, such as the MAC address and the IP address, Edge Switch A encapsulates the essential information into a connection creation message and sends it to Edge Switch B through the Retransmission Engine.
3) **Connection between the two hosts and Edge Switches.** When Edge Switch B receives the connection creation message, it uses the information from the message to forge an SYN message. This SYN message is almost the same as the one received by Edge Switch A. The only differences are the "sequence number" and "time-stamp". Edge Switch B uses its own sequence number and time-stamp in this SYN message. This SYN message will be sent to Host 2, and as the genuine TCP, Host 2 will reply an SYN-ACK message to "Host 1", but actually this packet will be forwarded to Edge Switch B.
4) **Imitate TCP interactions with the two hosts.** After Edge Switch B has replied an ACK message to the SYN-ACK message, a TCP connection is successfully established between Edge Switch B and Host 2. However, for Host 2, there is no difference in this design, as Host 2 believes it communicates with Host 1 using a TCP connection. If the TCP connection has been established correctly, Edge Switch B will send the information of Host 2 to Edge Switch A. Otherwise, Edge Switch B will send a connection failed message to

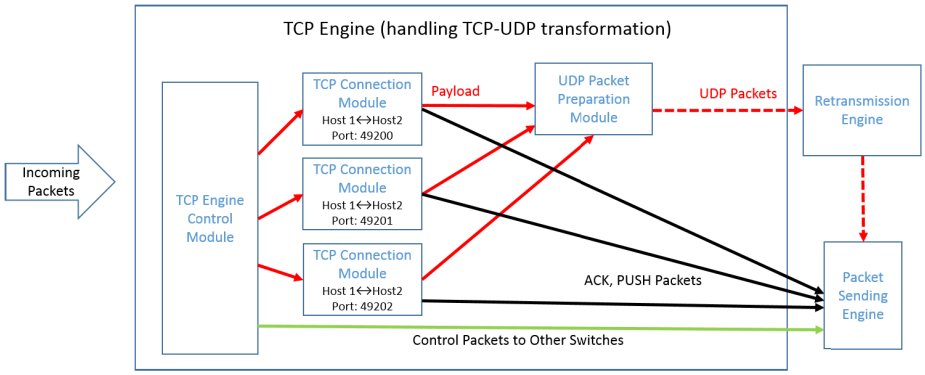

FIGURE 4. A scenario of TCP Engine in TCP-UDP transformation.

Edge Switch A. In the case of a success TCP connection establishment, Edge Switch A will receive a message from Edge Switch B, which includes the information about Host 2. Edge Switch A then uses the information except for the sequence number and the time-stamp, to forge an SYN-ACK message and send it to Host 1. Similar to the situation at Edge Switch B, a TCP connection now has been established between Host 1 and Edge Switch A. Similarly, in the perspective of Host 1, it believes that a TCP connection is established between Host 1 and Host 2. After the above procedure, an establishment of SDUDP connection has been completed between Host 1 and Host 2.

5) **Data transmission from Host 1 to the network.** Next, we address the transmission of actual data that Host 1 wants to send to Host 2. After the TCP handshake, Host 1 will send data to Host 2 through Edge Switch A, and Edge Switch A will keep acting like Host 2. Thus, Edge Switch A will buffer the data and send ACK messages back to Host 1. Interactions between Host 1 and Edge Switch A are the same as in a TCP connection. Edge Switch A will not buffer the data for a long time, because it will send this data to Edge Switch B. Once the data has been sent into the network, our framework guarantees the sequentiality and reliability through the switches with retransmission functions implemented in switches on the transmission path. To sum up, Edge Switch A sends the data and other switches on the route will make sure that the data arrives reliably at Edge Switch B in sequence and without loss. We describe the reliable mechanism transmission on the Retransmission Engine part in Section III-B2.
6) **Data receiving from the network to Host 2.** While the data arrives at Edge Switch B in sequence without any loss, Edge Switch B sends data to Host 2 according to the packet sequence via normal TCP mechanism.

The above is a typical procedure of data transmission from Host 1 to Host 2 under our framework. Benefiting from the centralized structure of SDN, in our design the controller saves detailed settings of the framework, including flow rules, buffer size on switches, and other related metrics. The controller exchanges messages and settings on switches via OpenFlow.

B. ENHANCED SDN SWITCH ARCHITECTURE

This section describes in detail the proposed major designs of the TCP Engine, the Retransmission Engine, and the Packet Sending Engine.

1) TCP ENGINE

The TCP Engine handles three main tasks: 1) maintenance of the TCP connections between hosts and Edge Switches; 2) decapsulation of TCP packets and encapsulation to UDP packets before sending to the network; 3) transforming UDP packets into TCP format before sending them to the destination host. As our design follows the current transmission protocol for the hosts, the TCP Engine maintains TCP connections with hosts. In addition, TCP Engines transform packet format during transmission in our framework. In our design, only Edge Switches need to be deployed in combination with the TCP Engine. In other words, switches other than Edge Switches can support our design only through the function of the Retransmission Engine. This design offloads the efforts of deploying the TCP engine in every switch. A scenario of TCP Engine in TCP-UDP transformation is shown in Figure 4. The TCP Engine in the figure is composed of 1) TCP Engine Control Module, 2) UDP Packet Preparation Module, and 3) TCP Connection Module. The following describes details about these components:

- **TCP Engine Control Module.** The TCP Engine Control Module handles the creation/termination of the TCP Connection Module and the communication with other switches in our framework. As all the TCP packets from a host will be delivered to the Edge Switch with TCP Engine, the control module retrieves these packets and analyzes the packet metadata. As in Figure 2, when a SYN message packet is sent from Host 1 and received by Edge Switch A, and the corresponding TCP connection is not established, the control module

creates a new TCP Connection Module for the TCP connection. The control module then encapsulates the essential information (e.g., the MAC address, the IP address, and the port number.) into a control message and send it to Edge Switch B which is nearest to the destination host (Host 2). Control modules also use control messages to negotiate an exclusive port for the TCP connection. As shown in Figure 2, the TCP connection of Host 1 and Host 2 uses port 49200. Consequently, once the control module in Edge Switch B receives packets that are sent to port 49200, it is able to identify that these packets are belonging to the connection of Host 1 and Host 2 and delivers them to the corresponding TCP Connection Module immediately.

On the other side, when the control module in Edge Switch B receives the control message from Edge Switch A, it creates a TCP Connection Module for Host 1 and Host 2. It starts sending SYN messages to Host 2 and waits for a SYN-ACK message. Once the TCP handshake has been completed, the control module sends a control message with essential information back to Edge Switch A.

As the control module in Edge Switch B delivers the essential information of Host 2 to the TCP Connection Module of Host 1 and Host 2 at Edge Switch A, the control module of Edge Switch A sends a SYN-ACK message to Host 1, completing the handshake procedure. Data from Host 1 is delivered to the TCP Connection Module of Host 1 and Host 2 at Edge Switch A. Up to now, the main task of the control module for establishing the connection has finished. It handles the creation of the TCP Connection Module and communication with other switches.

When terminating a connection, the corresponding TCP Connection Module notifies the control module to terminate the connection. Then the control module in Edge Switch A sends a control message to Edge Switch B to notify the termination of the TCP connection. The exclusive ports for Host 1 and Host 2 are also released.

- **TCP Connection Module.** The TCP Connection Module handles the TCP connection with the hosts. It guarantees that data is received intact from the host and is sent to the destination host properly. Also, the TCP Connection Module maintains essential information, such as the MAC address, the IP address and the port number of Host 1 and Host 2.

  The main tasks of the TCP Connection Module are data forwarding and interactions with hosts. While interacting with hosts, the TCP Connection Module has to implement complete and correct TCP mechanisms in order to guarantee the integrity of the data. Once the data is received correctly, the TCP Connection Module forwards the data to the Packet Sending Module for transmission to Edge Switch B.

  For UDP packets transmitted to Edge Switch B, the TCP Connection Module collects them according to their connection information; next, the TCP Connection Module transforms the orderly UDP packets into TCP format and send to the Packet Sending Module to forward the packets to the destination host (Host 2).

  In addition to the above-mentioned functions, to improve transmission efficiency, we have designed the **packet accumulation function** on it. Packets are delivered in PDU (Protocol Data Unit) [34] at the IP layer; even though the size of a packet at the IP layer could be greater than 1.5 KB, when it is delivered to the data link layer, the packet will be fragmented into smaller sizes that fit the MTU (Maximum Transmission Unit, generally 1500 bytes) [35] of a frame (PDU at data link layer). However, if the packet size is smaller than 1.5 KB, it will still be sent in a single frame. This means that, if the packet size is smaller than 1.5 KB, some space would remain underutilized in such frames. To solve this issue and improve transmission efficiency, in our framework, when sending packets, instead of sending a packet that is smaller than 1.5 KB, our framework accumulates data until its size exceeds 1.4 KB (about 0.1 KB are left for the header field), after which we send the packet. Through this design, we seek to optimize the utilization of the packets (frames) that are sent into the network.
- **UDP Packet Preparation Module.** As mentioned in the previous section, the UDP Packet Preparation Module receives packets that contain only UDP headers and data. In other words, the UDP Packet Preparation Module has to add the network layer and the data link layer headers for these packets. Since these UDP packets are used to transmit data between two Edge Switches, the switch's information of the network layer and the data link layer are added to headers.

  In this module, we use the identification fields at the network layer of IPv4 and IPv6 to help monitor the sequentiality of our transmission. In IPv4, the Identification field is a 16-bit and necessary field when forming packets. In IPv6, the Identification field is a 32-bit and optional field when forming packets. Identification is primarily used for reassembling and identifying the group of fragments of a single IP datagram. In our design, for the purpose of synchronizing switches with the TCP Engine in SDN, the initial value of the Identification field is set to zero. Then the value in the Identification field of the next frames in a single connection will increase by one. Once the middle switches find the value in the Identification field is not in order, the system considers there is a packet loss and starts the retransmission procedure as soon as possible.

  Once the packets are prepared, they will be sent to the Retransmission Engine. Note that when the Retransmission Engine has received complete UDP packets from the TCP Engine, they will notify the TCP Engine to clear the buffered packets to avoid large buffer requirement in the TCP Engine. In short, the UDP Packet Preparation Module's tasks are to aggregate packets from different

TCP connections and to add complete headers to them. Then the UDP Preparation Module delivers the packets to the Retransmission Engine.

2) RETRANSMISSION ENGINE

To guarantee that UDP packets are sent to switch B reliably before the packets are delivered to the network, these UDP packets need to be buffered for retransmission request if packet loss is detected. In our design, the Retransmission Engine handles this task.

As mentioned above, our framework detects whether packet loss occurs or not based on the Identification field at the IP layer meta field. Similar to the pre-defined flows in switches in our framework, the packet transmission path between Edge Switch A and Edge Switch B is fixed. If the identification values of incoming packets are sequential, there should be no packet loss, and the switch will notify the previous switch which is deployed with the Retransmission Engine to clear the buffer. Otherwise, packet loss should occur. When packet loss is detected, the Retransmission Engine identifies the identification values of the missing packets and then sends a retransmission request to the previous switch.

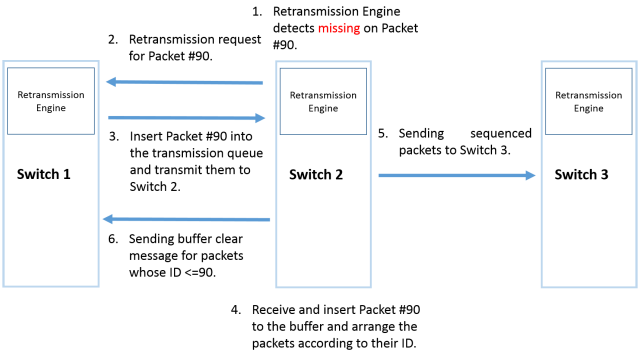

FIGURE 5. Retransmission procedure of Retransmission Engines in switches.

Figure 5 demonstrates a retransmission scenario. There are three switches with Retransmission Engines on the route between Edge Switch A and B, which are denoted as Switch 1, 2 and 3. Once Switch 2 detects a packet loss on Packet #90 (90 is the value in the Identification field), it sends a retransmission message (through UDP) to request Switch 1 to retransmit Packet #90 and start a timer. The timer is used for sending retransmission request messages periodically while waiting for retransmitted packets, because the message could also be lost on the way to Switch 1. Before receiving retransmitted packets from Switch 1, Switch 2 simply waits and buffers packets; it does not forward any packets to Switch 3. At the same time, when Switch 1 receives the retransmission request from Switch 2, it will retransmit the packets requested by Switch 2 as soon as possible. When Switch 2 has received all the missing packets, it will arrange these packets accordingly based on their Identification values and then forward them to Switch 3. Meanwhile, Switch 2 will notify Switch 1 to clear the requested and the previous packets (packet ID $<= 90$) in Switch 1's buffer to avoid large buffering on Retransmission Engines. Through this mechanism, the reliability during packet transmission is guaranteed. Note that since the deployment of the Retransmission Engine to every switch costs a certain amount of effort, the installing of the engine in every switch is optional. Further detailed evaluation results are described in Section IV-E.

3) PACKET SENDING ENGINE

The Packet Sending Module handles most of the packet transmissions in our design. The module aggregates packets from the TCP Engine Control Module, the TCP Connection Module, the UDP Packet Preparation Module, and the Retransmission Engine. After aggregation it sends all packets into the network.

C. IMPLEMENTATION NOTE

Our design uses Scapy [36] as the tool to achieve packet manipulation. In order to make Host 1 and Host 2 not observe any difference as the TCP connection under our infrastructure, the TCP Engine in Edge Switch A has to add headers that contain not only the IP address but also the MAC address of Host 2. To achieve this, Scapy uses raw sockets to manipulate and encapsulate packets.

IV. EVALUATION

We conducted our experiments on virtual machines of the Linux server with an Intel Core i7-6700 CPU, which is a quad-core processor at 3.40 GHz, and with 8 GB of physical memory. We used Linux version 3.19.0-25-generic, OpenFlow [37] version 1.0, Mininet [38] version 2.3.0, and Scapy version 2.2.0. All programs were implemented in Python, with Scapy library. In our experiments, Host 1 is the client to send data toward Host 2. They use TCP socket in Python to transmit files, a total size of 3,000 KB, at the rate of about 100 KB/sec.

We measure the performance using two important metrics in our experiments, including the number of packets used to transmit the file and the total number of bytes of these packets. From these two metrics, we compare the improvement of our framework with TCP connections.

A. EXPERIMENT SCENARIOS AND SETTINGS

In this experiment, one sample scenario (5 switches) is shown in Figure 6; Host 1 and Host 2 want to establish a TCP connection to transmit data. Edge Switch A and B are deployed with TCP Engines and Retransmission Engines. Other switches are deployed with Retransmission Engines only. We set up 4 different scales of network, respectively consisting of 5, 7, 9, 11, and 13 switches. Among the SDN, each link has a 20 ms delay and a designated loss rate. To imitate different network conditions, our experiment set up five settings of loss rate, 0.00%, 0.25%, 0.50%, 0.75% and 1.00%. For each experiment, we set a fixed loss rate to every link in the network.

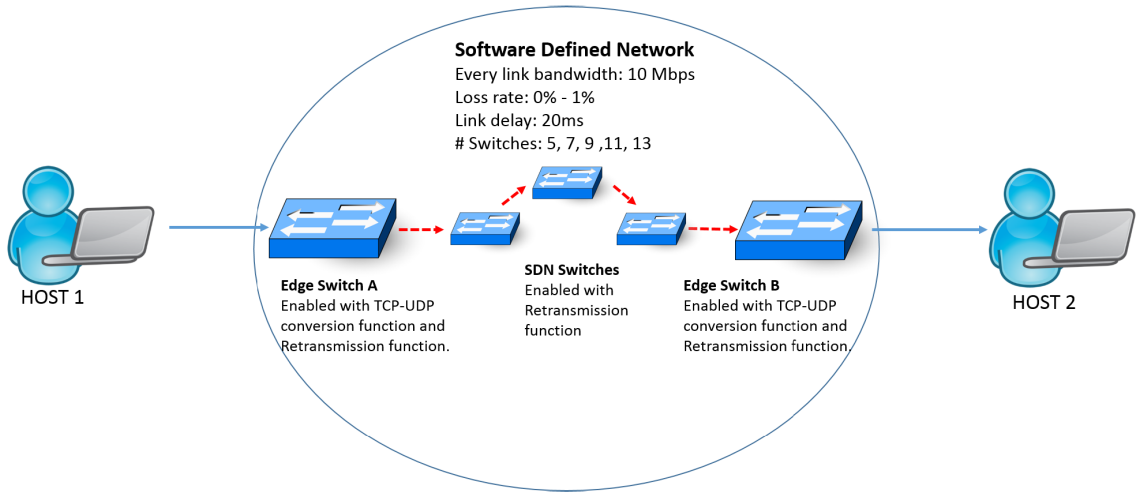

FIGURE 6. Scenario of the experiment (5 switches).

TABLE 1. 5 filesets used for experiment.

Fileset	size of file	# files in set
A	0.5KB	6000
B	1KB	3000
C	10KB	300
D	100KB	30
E	1000KB	3

Next, we investigate whether different file sizes will influence the load of switches either in TCP transmission and under our framework. In this experiment, Host 1 sends a 4 different filesets to Host 2, as shown in Table 1. The size of each fileset is 3,000 KB in total and consists of a different number of files with different sizes. We set the file sizes at 0.5KB, 1KB, 10KB, 100KB or 1,000KB to validate the performance of our design in different data transmission scenarios.

In every combination of settings, we conduct an experiment of data transmission via our proposed framework and via TCP. The results are shown in Figure 7 and Figure 8. The experiment results conducted using our framework are shown in solid lines, and those under TCP in dot-dash lines. The different colors of the lines represent the different file sets we used. We conduct every experiment for 10 rounds and present the average value and 95% confidence interval in the figure. Figure 7 presents the average load of each switch in terms of number of bytes; Figure 8 presents the average load of each switch in terms of packets. In the normal TCP connection scenario, we use normal switches without TCP Engines and Retransmission Engines, while the two Engines are enabled when conducting the experiment under our framework.

B. OVERALL COMPARISON WITH TRADITIONAL TCP

Figure 7 and Figure 8 present the results and shows that our proposal significantly outperforms the TCP transmission in both bytes and packets used for transmission with every fileset and link loss rate. In Figure 9, we demonstrate that SDUDP significantly outperforms TCP while the link delay is set to 1 ms (9 switches). From the results, the average load of each switch in SDUDP smoothly increases with the rising loss rate. On the contrary, the TCP transmission requires a larger number of bytes while the loss rates rise. Figure 8 shows that transmission via the TCP protocol requires much more packets while the file size is small (0.5KB). Since one TCP packet normally carries about 1400 bytes of data, when the file size is small as in file set A, we observe that some TCP packet carry merely 500 bytes of data. Therefore, to complete the data transmission, transmitting data through TCP requires more packets and bytes than through our design.

A noticeable phenomenon that occurs in TCP transmission is that the average load of each switch descends with the rising loss rate. To figure out the reason, we use Wireshark [39] to observe the condition of packets during data transmission. Our observations show that when loss rate is 0.00%, most of the packets are at an average size of about 1,500 bytes. However, when the loss rate rises, Host 2 sends TCP window update messages to Host 1 for upgrading its window size to bigger ones after packet losses occur. We did not observe any TCP window update message when the loss rate was 0.00%. However, after the TCP window updates messages, Host 1 begins to send some packets at the size about 3,000 bytes since it realizes that Host 2's window size is large enough for buffering packets. Thus, when there is a higher loss rate, Host 1 transmits data more efficient through larger packets. In other words, the transmission is finished using fewer packets and that is one possible reason that why the average load (in terms of packets) of switches descends as the loss rate rises. Nevertheless, our framework still outperforms TCP in all aspects.

C. DIFFERENT NUMBER OF SWITCHES

Figure 7 and Figure 8 present our design's experiment results and compares them to TCP in different network

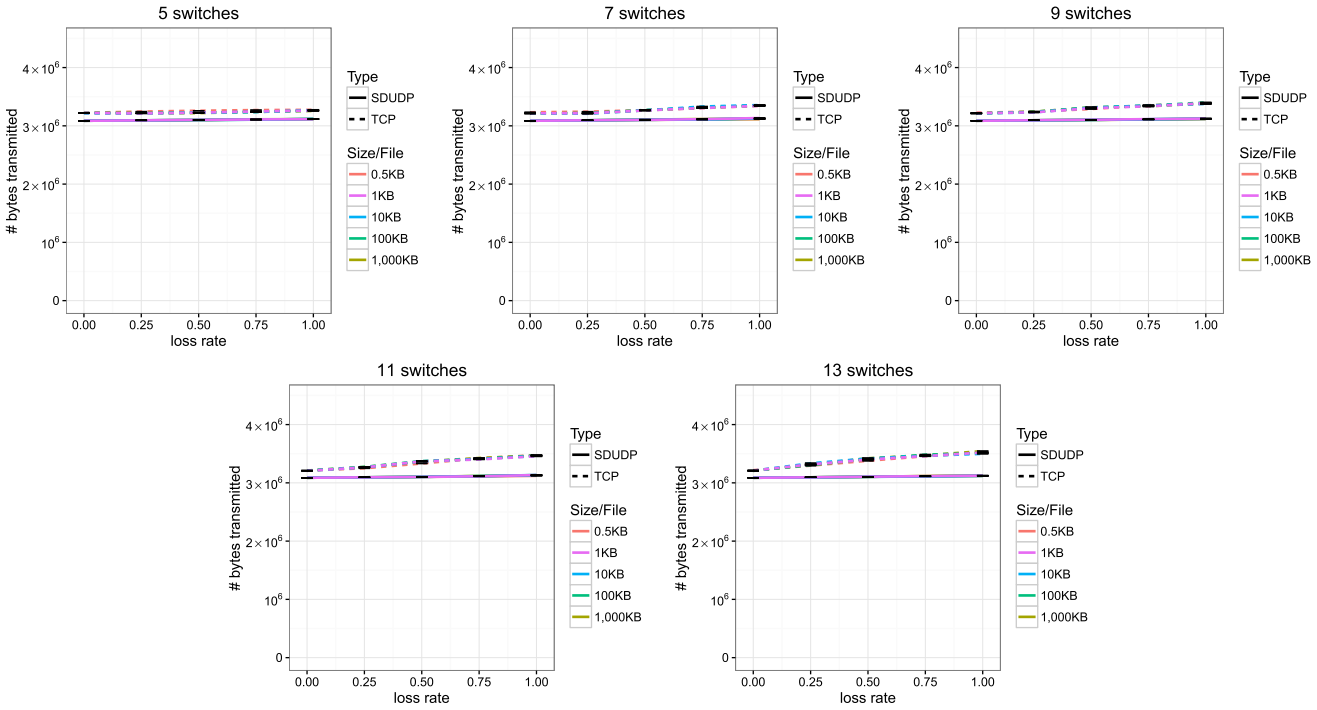

FIGURE 7. Average load (in terms of bytes) of different numbers of switches.

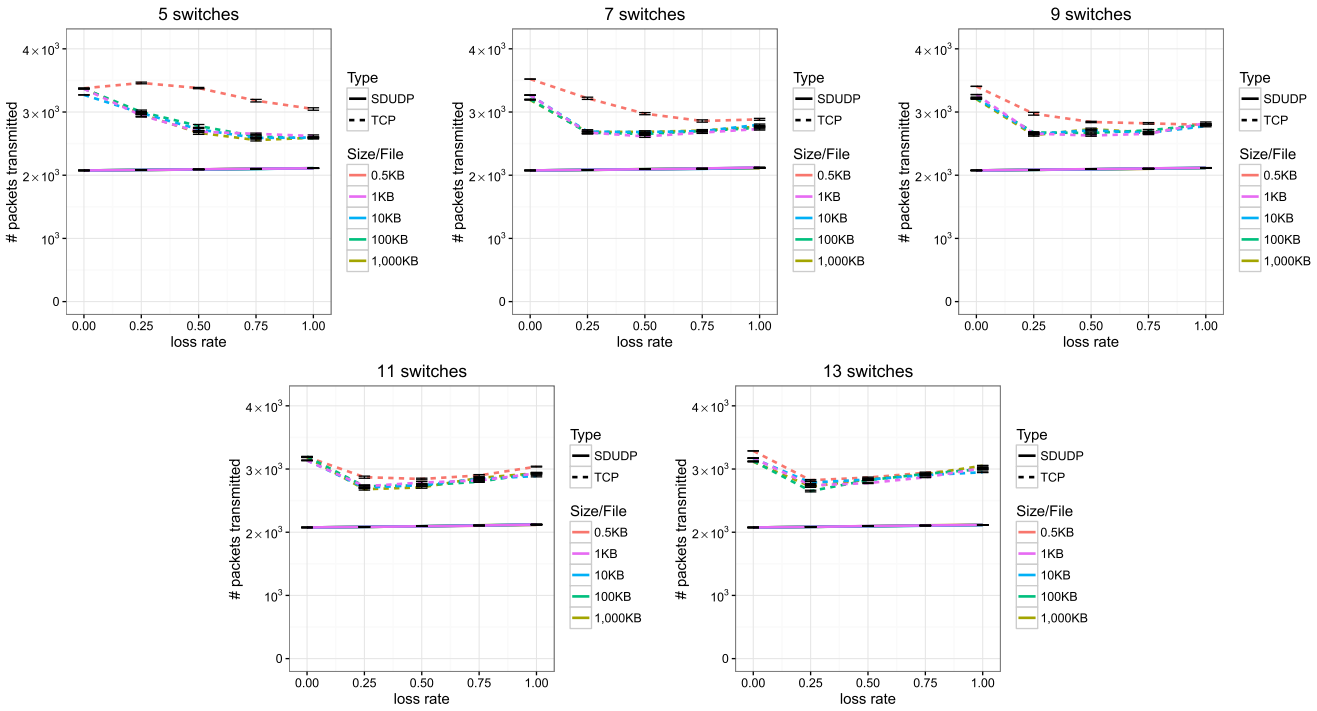

FIGURE 8. Average load (in terms of packets) of different numbers of switches.

environments. Each sub-figure indicates the load under different numbers of switches. From the results shown in Figure 7, the average load increases with more switches between Host 1 and Host 2, as in Table 2. The results could be attributed to the fact that the host-to-host loss rate becomes higher when the number of links between Host 1 and Host 2

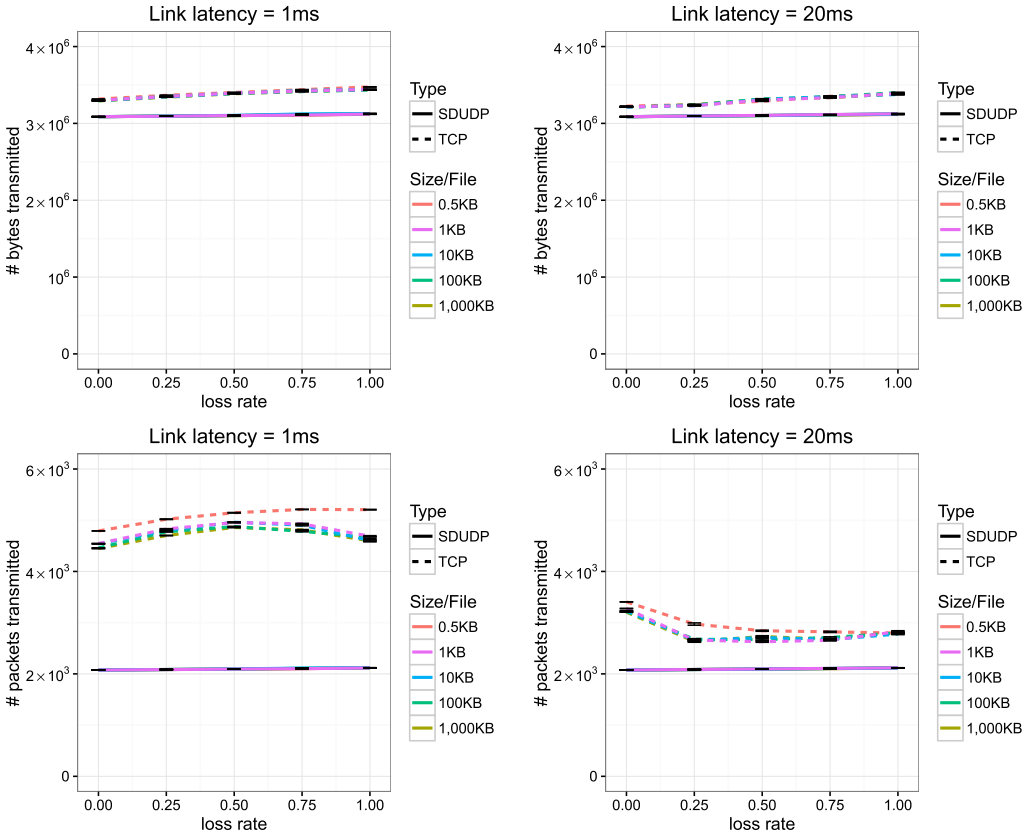

FIGURE 9. Average load of different numbers of switches in different link delay (l ms and 20 ms, 9 switches).

TABLE 2. Host-to-host loss rates of different link loss rates and different switches of the path (calculated by inclusion-exclusion principle).

LR per Link	5 S	7 S	9 S	11 S	13 S
0.00 %	0.000 %	0.000 %	0.000 %	0.000 %	0.000 %
0.25 %	0.996 %	1.491 %	1.983 %	2.472 %	2.959 %
0.50 %	1.985 %	2.963 %	3.931 %	4.889 %	5.838 %
0.75 %	2.966 %	4.416 %	5.845 %	7.252 %	8.638 %
1.00 %	3.940 %	5.852 %	7.726 %	9.562 %	11.362 %

increases. Thus, packets loss and retransmissions happen more frequently while the scale of the network becomes larger.

From Figure 8, as described in the previous section, we may observe that when the loss rate rises, normal TCP transmits data with larger packets. Therefore, our system achieves a larger improvement at loss rate of 0.00% than at other loss rates. A noticeable phenomenon is that the average load (in terms of packets) gradually increases when the loss rates come higher than 0.25% with seven, nine, and eleven switches. The reason is most likely that as the number of switches increases, the actual loss rate between Host 1 and Host 2 also increases. In other words, TCP transmission sends more packets for retransmission and acknowledgment.

D. EFFECTS OF PACKET ACCUMULATION FUNCTION

In SDUDP we design the packet accumulation function to improve the utilization of the packets that are sent into the network. To evaluate the advantage of the design, we conduct another experiment to compare the packet usage of transmitting files in TCP, in SDUDP with/without the packet accumulation function, under a 5-switch network scenario.

Figure 10 demonstrates the packet usage while transmitting under TCP, SDUDP with the packet accumulation function, and SDUDP without the packet accumulation function. From the results, the packet accumulation function benefits SDUDP to reduce more than 20% of packet usage while transmitting 0.5 KB files. For larger files, the improvement is about 11-14%. Furthermore, SDUDP still outperforms TCP even if the packet accumulation function is disabled. Considering that more packets would consume an increasing amount of the computing and bandwidth resource for header fields and packet encapsulation, in SDUDP, we use the packet accumulation function to decrease such overhead on switches and engines.

E. PLACEMENT OF SWITCHES WITH THE RETRANSMISSION ENGINE

In previous experiments, we deployed the Retransmission Engine in every switch on the network. However, deploying such a function on every switch is time and effort consuming. To understand the effectiveness of the deployment on the loading of switches, we conducted another

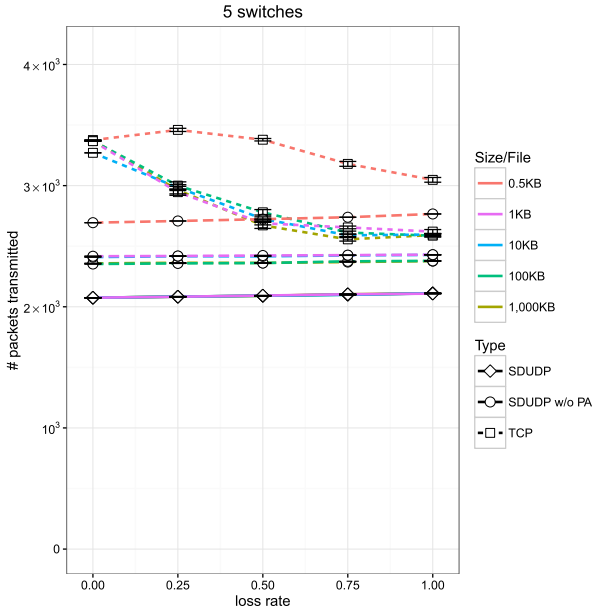

FIGURE 10. Packet transmitted in TCP, SDUDP with the packet accumulation function, and SDUDP without the packet accumulation function (5 switches).

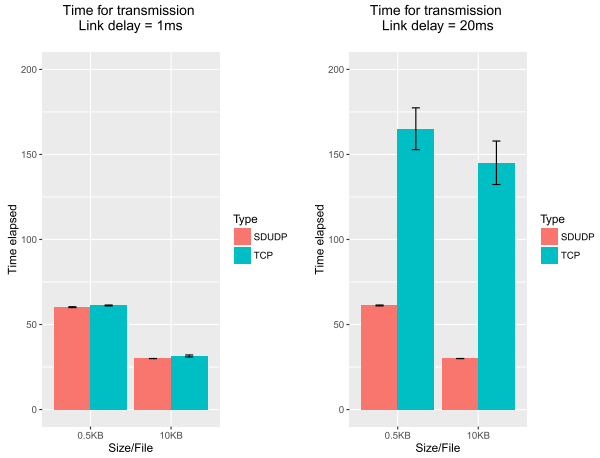

FIGURE 11. Time consumption for transmitting 3,000KB file through TCP and SDUDP (with different link delay).

experiment to investigate the performance of partially deploying Retransmission Engines on switches.

In this experiment, there are nine switches in total (including Edge Switch A and Edge Switch B), and the switches are labeled from 1 to 9; switch 1 is Edge Switch A, and switch 9 is Edge Switch B. There are three scenarios in this experiments, including 1) deploying Retransmission Engines on all the switches, 2) on odd-numbered switches, and 3) on the 1st, 5th, and 9th switches. In these experiments, we use a fileset D which contains 30 files; each file is 100 KB.

From the results, when we deploy fewer switches with the Retransmission Engine, the average load of each of the nine switches increases at about 1.1-1.3% (link loss rate = 1.0%). Let us take case 1 (Retransmission Engine on every switch) and 3 (Retransmission Engine on the 1st, 5th, 9th switch) as an example. In case 1, the retransmission requests and retransmitted packets are transmitted through two neighbor switches. For example, when packets loss occurs between switch 2 and 3, retransmission requests and retransmitted packets occur only between these two switches at a loss rate of 1.00%. In case 3, retransmission requests and retransmitted packets exist between switch 1 and switch 5 at a loss rate of 3.94%. In other words, switch 1 to 5 have to suffer the load of retransmitting packets at a higher loss rate. That is the reason why switches of case 3 require more packet/byte loading than in case 1.

In a practical scenario, deploying fewer switches with the Retransmission Engine implies lower cost; however, the average load of individual switches will slightly increase. Therefore, the trade-off between cost and load of switches should be determined. For example, deployment of more switches with a Retransmission Engine can be considered on a network in a bad environment; the contrary could be an option in a more stable network environment, such as a wired network.

F. TIME CONSUMPTION

According to our design, SDUDP avoids long waiting time for retransmissions comparing with TCP. To evaluate the advantage of the fast retransmission mechanism in SDUDP, we measure the total time for transmitting 3,000 KB files (0.5 KB and 10 KB for each file). We set the setting of the SDN as follows: 10 Mbps for the link bandwidth; 9 switches for the transmission route; 1 ms and 20 ms for the link delay between switches; and 1% for the link loss rate.

As the results shown in Figure 11, we find that SDUDP could significantly reduce the time consumption of transmitting files while the network latency is high. The results imply that benefiting from the fast retransmission mechanism implemented in the retransmission engines of SDUDP, our design is more suitable for the network environments with high latency or encountering more transmission errors.

G. SUMMARY

To sum up, our proposed framework achieves great improvement in comparison to normal TCP. When the loss rate rises, TCP transmission suffers much more overhead than our framework owing to the high latency and inefficient retransmission mechanisms in TCP. Our framework, contrarily, provides an efficient retransmission mechanism. When it comes to the placement of switches with Retransmission Engine, loss rate should be considered to achieve a balance between cost and load of switches.

V. CONCLUSION AND FUTURE WORKS

This paper proposes a novel reliable transmission framework. In comparison to TCP transmissions, our design leverages UDP packets to offload the transmission overhead of TCP headers and acknowledgments. In addition, we leverage flow

rules on SDN to ensure that packets under our proposed framework will pass through predefined routes, in the process achieving transmission reliability through Retransmission (if packets are lost) Engines. Furthermore, in order to utilize each packet (frame) sent in to the network to the fullest, our framework accumulates packets (data) to better utilize the bandwidth, especially when transmitting a large number of small packets. Due to this mechanism, we have measured at least a 10%-30% improvement in the aspect of average load (in terms of packets) of switches.

The contributions of this study can be summarized as follows:

1) Packets are transmitted in the primitive UDP format, which contains no sequence numbers or other complex transport layer header fields. Consequently, the overhead of the long header is reduced.
2) Reliability of packets is guaranteed by our proposed engines deployed in SDN switches. ACK messages are not needed in our framework, which means that the bandwidth for ACK messages is saved.
3) Our design accumulates underutilized smaller packets before sending them, reducing the amount of total packets required while transmitting small files over the network.
4) Our design leverages the power of SDN to detect packet loss events according to Identification field values. By the design, a retransmission procedure could start as soon as possible rather than waiting for a timeout as in TCP.

Although our proposed framework is based on SDN, our framework can easily be deployed to a larger network, e.g., a campus network. This is because the engines in switches can exist independent as a node in WAN. The edge node (TCP engine) aggregates TCP connections and transmits data to middle nodes with Retransmission Engines. Furthermore, our design is especially suitable for a network with high latency and for real-time applications while the network transmission errors often occur, as we significantly reduce the required time for starting retransmissions. In SDUDP, it can be checked whether packets were lost on the way of transmission instead of on arrival at the destination host. Once packets are lost, retransmission can take place immediately by middle nodes with engines rather than by hosts through waiting for TCP timeouts or receiving duplicate ACK messages.

However, some issues remain to be addressed to improve the capability and availability of our design. 1) Porting on other platforms. To extend the use of our design, methods need to be found to implement the framework on other platforms such as Open VSwitch's kernel datapath or OpenFlow Data Plane Abstraction [40]. 2) The TCP-UDP packet transformation can be accelarated. In our current implementation, packet manipulation takes place at the application layer, after which data is sent directly at data link layer through Scapy (raw socket). However, while the number of packets increases, the overhead may degrade switches performance. We consider if the transformation process of the packet format from TCP to UDP can be performed and accelerated by a hardware device, it would be more efficient and reliable. 3) Handling TLS in our design. As our design uses UDP packets to transmit data over the network, how to guarantee the security and meet the standard of TLS is a worthy question of investigation left for future research. 4) Evaluation of SDUDP on physical devices. Even though we have conducted an experiment on Mininet, many practical issues such as the packet transformation overhead and buffering size on switches or engines may downgrade the system performance. In addition, while the network size becomes larger, the scalability of our framework may be another issue. Implementing SDUDP on physical devices and conducting evaluations at different scales of networks to investigate the system performance are worthy of being studied in the future.

REFERENCES

[1] *RFC 793—Transmission Control Protocol*, accessed on Apr. 1, 2017. [Online]. Available: https://tools.ietf.org/html/rfc793

[2] *Rfc 768—User Datagram Protocol*, accessed on Apr. 1, 2017. [Online]. Available: https://tools.ietf.org/html/rfc768

[3] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[4] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2013, pp. 1–13.

[5] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: An intellectual history of programmable networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, 2014.

[6] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using OpenFlow: A survey," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 1, pp. 493–512, 1st Quart., 2014.

[7] T. Luo, H.-P. Tan, and T. Q. S. Quek, "Sensor OpenFlow: Enabling software-defined wireless sensor networks," *Commun. Lett.*, vol. 16, no. 11, pp. 1896–1899, Nov. 2012.

[8] S. Costanzo, L. Galluccio, G. Morabito, and S. Palazzo, "Software defined wireless networks: Unbridling SDNs," in *Proc. Eur. Workshop Softw. Defined Netw. (EWSDN)*, Oct. 2012, pp. 1–6.

[9] *Draft-Ietf-Sigtran-Reliable-UDP-00—RELIABLE UDP PROTOCOL*, accessed on Apr. 1, 2017. [Online]. Available: https://tools.ietf.org/html/draft-ietf-sigtran-reliable-udp-00

[10] *RFC 4347—Datagram Transport Layer Security*, accessed on Apr. 1, 2017. [Online]. Available: https://tools.ietf.org/html/rfc4347

[11] *RFC 6347—Datagram Transport Layer Security Version 1.2*, accessed on Apr. 1, 2017. [Online]. Available: https://tools.ietf.org/html/rfc6347

[12] D. McGrew and E. Rescorla, "Datagram transport layer security (DTLS) extension to establish keys for secure real-time transport protocol (SRTP)," Internet Eng. Task Force (IETF), Fremont, CA, USA, Tech. Rep. RFC 5764, 2010.

[13] E. He, J. Leigh, O. Yu, and T. A. DeFanti, "Reliable blast UDP: Predictable high performance bulk data transfer," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2002, pp. 317–324.

[14] Y. Gu and R. Grossman, "SABUL: A transport protocol for grid computing," *J. Grid Comput.*, vol. 1, no. 4, pp. 377–386, 2003.

[15] M. R. Meiss, "Tsunami: A high-speed rate-controlled protocol for file transfer," Indiana Univ., Bloomington, IN, USA, 2004.

[16] Y. Gu and R. L. Grossman, "UDT: UDP-based data transfer for high-speed wide area networks," *Comput. Netw.*, vol. 51, no. 7, pp. 1777–1799, 2007.

[17] A. O. F. Atya and J. Kuang, "RUFC: A flexible framework for reliable UDP with flow control," in *Proc. 8th Int. Conf. Internet Technol. Secured Trans. (ICITST)*, Dec. 2013, pp. 276–281.

[18] S. Kopparty, S. V. Krishnamurthy, M. Faloutsos, and S. K. Tripathi, "Split TCP for mobile ad hoc networks," in *Proc. Global Telecommun. Conf. (GLOBECOM)*, vol. 1. Nov. 2002, pp. 138–142.

[19] M. Luglio, M. Y. Sanadidi, M. Gerla, and J. Stepanek, "On-board satellite 'split TCP' proxy," *IEEE J. Sel. Areas Commun.*, vol. 22, no. 2, pp. 362–370, Feb. 2004.
[20] V. Farkas, B. Héder, and S. Nováczki, "A split connection TCP proxy in LTE networks," in *Meeting of the European Network of Universities and Companies in Information and Communication Engineering*. Berlin, Germany: Springer, 2012, pp. 263–274.
[21] M. Ivanovich, P. W. Bickerdike, and J. C. Li, "On TCP performance enhancing proxies in a wireless environment," *IEEE Commun. Mag.*, vol. 46, no. 9, pp. 76–83, Sep. 2008.
[22] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Commun. Mag.*, vol. 51, no. 2, pp. 114–119, Feb. 2013.
[23] A. Gudipati, D. Perry, L. E. Li, and S. Katti, "SoftRAN: Software defined radio access network," in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw.*, 2013, pp. 25–30.
[24] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 3, pp. 1617–1634, 3rd Quart., 2014.
[25] D. Kreutz, F. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.
[26] J. Li, M. Veeraraghavan, S. Emmerson, and R. D. Russell, "File multicast transport protocol (FMTP)," in *Proc. 15th IEEE/ACM Int. Symp. Cluster, Cloud Grid Computing (CCGrid)*, May 2015, pp. 1037–1046.
[27] X. Ji, Y. Liang, M. Veeraraghavan, and S. Emmerson, "File-stream distribution application on software-defined networks (SDN)," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 2. Jul. 2015, pp. 377–386.
[28] S. Chen, M. Veeraraghavan, S. Emmerson, J. Slezak, and S. G. Decker, "A cross-layer multicast-push unicast-pull (MPUP) architecture for reliable file-stream distribution," in *Proc. IEEE 40th Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, Jun. 2016, pp. 535–544.
[29] P. J. Frantz and G. O. Thompson, "VLAN frame format," U.S. Patent 5 959 990 A, Sep. 28, 1999
[30] *RFC 791—Internet Protocol*, accessed on Apr. 1, 2017. [Online]. Available: https://tools.ietf.org/html/rfc791
[31] *RFC 2460—Internet Protocol, Version 6 (IPv6) Specification*, accessed on Apr. 1, 2017. [Online]. Available: https://tools.ietf.org/html/rfc2460
[32] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about SDN flow tables," in *Proc. Int. Conf. Passive Active Netw. Meas.*, 2015, pp. 347–359.
[33] *OpenFlow—Enabling Innovation in Your Network*, accessed on Jul. 2, 2017. [Online]. Available: http://archive.openflow.org/
[34] *Protocol Data Unit—Wikipedia*, accessed on Apr. 1, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Protocol_data_unit
[35] *Maximum Transmission Unit—Wikipedia*, accessed on Apr. 1, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Maximum_transmission_unit
[36] *Scapy*, accessed on Apr. 1, 2017. [Online]. Available: http://www.secdev.org/projects/scapy/
[37] *OpenFlow—Open Networking Foundation*, accessed on Apr. 1, 2017. [Online]. Available: https://www.opennetworking.org/sdn-resources/openflow
[38] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proc. 9th ACM SIGCOMM Workshop Hot Topics Netw.*, 2010, p. 19.
[39] *Wireshark Download*, accessed on Apr. 1, 2017. [Online]. Available: https://www.wireshark.org/download.html
[40] *OF-DPA Software*, accessed on Apr. 1, 2017. [Online]. Available: https://www.broadcom.com/products/ethernet-connectivity/software/of-dpa

MING-HUNG WANG (S'17) received the B.S. degree in computer science and the M.S. degree in communication engineering from the National Tsing-Hua University, in 2008 and 2010, respectively. He is currently pursuing the Ph.D. degree with the Department of Electrical Engineering, National Taiwan University. His research interests include network security, social media analysis, and software-defined networking. His works have been published in the IEEE INFOCOM, the IEEE TrustCom, the IEEE CCNC, and the IEEE SMC conferences.

LUNG-WEN CHEN received the B.S. degree in computer science from National Tsing-Hua University in 2014 and the M.S. degree in electrical engineering from National Taiwan University in 2016. His research interests include computer networking and distributed systems.

PO-WEN CHI (M'17) received the B.S., M.S., and Ph.D. degrees from National Taiwan University in 2003, 2005, and 2015, respectively, all in electrical engineering. He is currently a Senior Engineer with Arcadyan Technology Corporation. His research interests include network security, applied cryptography, software-defined networking, and telecommunications.

CHIN-LAUNG LEI received the B.S. degree in electrical engineering from National Taiwan University, Taipei, in 1980, and the Ph.D. degree in computer science from The University of Texas at Austin in 1986. From 1986 to 1988, he was an Assistant Professor with the Computer and Information Science Department, The Ohio State University, Columbus. In 1988, he joined the Faculty of the Department of Electrical Engineering, National Taiwan University, where he is currently a Professor. He has authored over 250 technical articles in scientific journals and conference proceedings. His current research interests include network security, cloud computing, and multimedia QoE management. He was a co-recipient of the First IEEE LICS Test-of-Time Award.