PAPER

Firewall Traversal Method by Pseudo-TCP Encapsulation

Keigo TAGA[†*], ***Nonmember***, **Junjun ZHENG**[†], **Koichi MOURI**[†], **Shoichi SAITO**[††], ***and*** **Eiji TAKIMOTO**[†††a)], ***Members***

SUMMARY A wide range of communication protocols has recently been developed to address service diversification. At the same time, firewalls (FWs) are installed at the boundaries between internal networks, such as those owned by companies and homes, and the Internet. In general, FWs are configured as whitelists and release only the port corresponding to the service to be used and block communication from other ports. In a previous study, we proposed a method for traversing a FW and enabling communication by inserting a pseudo-transmission control protocol (TCP) header imitating HTTPS into a packet, which normally would be blocked by the FW. In that study, we confirmed the efficiency of the proposed method via its implementation and experiments. Even though common encapsulating techniques work on end-nodes, the previous implementation worked on the relay node assuming a router. Further, middleboxes, which overwrite L3 and L4 headers on the Internet, need to be taken into consideration. Accordingly, we re-implemented the proposed method into an end-node and added a feature countering a typical middlebox, i.e., NAPT, into our implementation. In this paper, we describe the functional confirmation and performance evaluations of both versions of the proposed method.

key words: *QUIC, encapsulation*

1. Introduction

A wide range of communication protocols has recently been developed to address service diversification. Improvements in quality of service and communication performance can be expected when using these communication protocols. At the same time, firewalls (FWs) are installed at network boundaries to improve security. FWs filter packets based on their settings. To ensure security, FWs are set in a whitelist format that allows only the minimum amount of communication to be used by the nodes under the FW and blocks all other communication. Therefore, many FWs allow communication only to minimum protocols that are widely used in the Internet, such as Hypertext Transfer Protocol (HTTP), HTTP Secure (HTTPS), and Domain Name System (DNS). Security is ensured by the FW; however, there is a problem in that communication with developed protocols is blocked by the FW. In other words, even if a service provider attempts to deploy a service using a newly developed protocol and the end-terminal supports the new protocol, a situation may occur in which the new protocol cannot be used due to the FW. In other words, an FW may become a barrier preventing the spread of new protocols and improvements in service quality. In such a case, this problem can be solved by changing the FW settings. However, not all users have the authority to change FW settings. In addition, changing the FW configuration is not preferable from the viewpoint of network management. In particular, from the viewpoint of security, it is necessary to carefully determine protocols that cannot manage the state using a user diagram protocol (UDP) and protocols that dynamically change the port to be used.

If the FW settings cannot be changed, tunneling with protocols that are allowed by the FW such as HTTP and HTTPS can be effective as a method for passing through the FW. However, because HTTP and HTTPS work on transmission control protocol (TCP), the TCP control under HTTP tunneling or HTTPS tunneling will affect the performance of the tunneled communication. For this reason, a UDP-based tunneling protocol is preferred to take advantage of the characteristics of tunneled communication. DNS tunneling is a UDP-based tunneling method. However, FWs may filter DNS communication based on the Internet Protocol (IP) address. In addition, UDP-based DNS has a data limit of 512 bytes.

Therefore, there are various limitations to DNS tunneling. From the above, non-controlling UDP-based tunneling is preferable when considering the communication performance, however, TCP-based tunneling must be used when considering FWs.

Therefore, we proposed an encapsulation technology using pseudo-TCP to achieve FW traversal without losing the characteristics of the transport layer protocol [1], [2]. The proposed method encapsulates packets created at the transport layer such as UDP packets. Encapsulation is performed with a TCP header that specifies port 443, which is used in HTTPS as the destination port. Consequently, the FW recognizes the packet as an HTTPS packet and allows it to pass through the FW. The proposed method, pseudo-TCP, is also effective for stateful inspection FW because it emulates three-way handshake, sequence number, and acknowledgment control. The proposed pseudo-TCP does not provide various controls such as flow control, retransmission control, and congestion control; therefore it can communi-

Manuscript received March 6, 2021.
Manuscript revised August 2, 2021.
Manuscript publicized September 29, 2021.

[†]The authors are with the Ritsumeikan University, Kusatsu-shi, 525–8577 Japan.
[††]The author is with the Nagoya Institute of Technorolgy, Nagoya-shi, 466–8555 Japan.
[†††]The author is with the Hiroshima Institute of Technology, Hiroshima-shi, 731–5193 Japan.
[*]Presently, the author is with the NDR Co., Ltd.
a) E-mail: e.takimoto.k5@it-hiroshima.ac.jp

DOI: 10.1587/transinf.2021EDP7050

cate without degrading the characteristics of the transport layer protocol of the encapsulated communication.

In previous studies, we implemented the proposed method in QUIC [3] and performed experiments to comfirm the effectiveness of the proposed method. In the these studies, the implementation was that of a relay node, not an end-node for communication.

However, many technologies that use encapsulation, such as tunneling technology, perform encapsulation at the end nodes. Therefore, the proposed method was also implemented on the end-node. In addition, when applying the proposed method, there is a concern that the communication of the proposed method may fail due to a middlebox on the communication path. Therefore, the proposed method was modified to support Network Address Port Translation (NAPT), which is a typical middlebox.

The rest of this paper is organized as follows. An overview of related studies is provided in Sect. 2. The proposed FW traversal method that inserts a pseudo-TCP header is presented in Sect. 3. The application of the proposed method to QUIC and its evaluation are described Sects. 4 and 5, respectively. Conclusions and future work are detailed in Sect. 6.

2. Related Works

Tunneling is a technique for connecting two points on networks that are physically or logically separated by a virtual line. In addition, tunneling provides communication protocols blocked by FWs to communicate over the Internet. In the general tunneling methods, tunneling protocols encapsulate packets of the target protocol by other protocols. Therefore, the packets are handled as packets of the tunneling protocol on the Internet.

There has been a lot of studies and developments on tunneling and encapsulation. These technologies can be categorized by the layer at which encapsulation works.

In recent, HTTPS, which is an application layer protocol, has been widely used for encapsulation because HTTPS encrypts a payload by TLS. Therefore, HTTPS encapsulation makes insecure protocols, such as DNS, secure with TLS. HTTPS is also used for FW traversal since FWs allow HTTPS. However, HTTPS encapsulation increases in processing time deriving from that HTTPS is the application layer protocol. In addition, HTTPS uses TCP as a transport layer protocol, but the communication control of TCP conflicts with the communication control of the encapsulated target protocol [4], [5]. The confliction has a negative influence on the expected communication performance.

SoftEther [6] encapsulates Ethernet frames. SoftEther uses HTTPS as the tunneling protocol; therefore, the encapsulated packet is encrypted and passes through an FW that checks the packet in detail up to the application layer. However, a TCP connection is used for the transmission. As a result, when TCP is used as the L4 protocol for the tunneled communication, TCP double control occurs and the performance is degraded when a retransmission occurs. In addition, when a protocol other than TCP is used as the L4 protocol, the control of the tunneled communication is affected by the TCP control. As a result, communication with the original protocol becomes possible. SoftEther attempts to improve throughput by multiplexing TCP connections; however, it cannot achieve a speed that occupies the line bandwidth up to the capacity limit of the physical Internet line. Further, because TCP operates in the lower layer of the tunneling protocol, redundant control occurs. In addition, SoftEther has a UDP acceleration mechanism that uses HTTPS communication just to stay alive after the connection is established, and other data are transmitted by UDP to further speed up the communication. To achieve UDP communication, endpoints behind the NAPT/FW performs UDP hole punching. However, because of the source/destination port number is dynamically determined, this mechanism cannot be used from end-nodes under severely restricted FWs.

Generic Routing Encapsulation (GRE) [7] and IPsec [8] are tunneling protocols that encapsulate a network layer protocol with IP. Since the protocol type of GRE is different from TCP/UDP, FWs and middleboxes, especially NAPT, may block GRE packets. Furthermore, GRE has no encryption function. IPsec encapsulation is superior in terms of confidentiality because it encrypts IP packets as the payload. IPsec is affected by NAPT as well as GRE. For both IPsec and GRE, NAT/NAPT traversal which encapsulates the payload in UDP has been proposed [9], [10]. However, UDP encapsulation does not affect FWs that block UDP communication except DNS.

Point to Point Tunneling Protocol (PPTP) [11], Layer 2 Forwarding (L2F) [12], and Layer 2 Tunneling Protocol (L2TP) [13] are tunneling protocols that encapsulate L2 Point to Point Protocol (PPP) frames. PPTP has an encryption function but uses GRE for encapsulation. That is, PPTP has the same drawback as GRE. L2F and L2TP encapsulate the payload in UDP thus they can not traverse FWs. They do not have an encryption function, therefore they generally use IPsec in conjunction.

Recently, tunneling protocols for network virtualization have been proposed. Network Virtualization using Generic Routing Protocol (NVGRE) [14] and Virtual Extensible LAN (VXLAN) [15] encapsulate L2 frames with L3; NVGRE uses GRE for encapsulation, and VXLAN encapsulates L2 frames with UDP. Thus, they include the shortcomings of UDP encapsulation and GRE described above. Furthermore, some studies revealed their performance problem. Stateless Transport Tunneling Protocol (STT) [16] is an L2 tunneling protocol that does not affect the control of the tunneled communication. STT uses pseudo-TCP to make it possible to use TCP Segmentation Offload (TSO) and Large Receive Offload (LRO), which are offload mechanisms installed in the network interface card (NIC), and to achieve higher throughput and a lower CPU load than other tunneling protocols. However, STT uses some fields in the pseudo-TCP header for its own use. In addition, because there is no state, the control flag unchanges. Therefore, the middlebox managing TCP state discards STT packets as invalid TCP packets. Therefore, STT cannot be used for FW traversal.

Transparent Transport Tunneling (T3) [17] is a hybrid tunneling for virtual networks. T3 encapsulates an L2 frame by TCP if the frame includes a TCP packet. If the frame includes a UDP packet, T3 encapsulates it by UDP. A tunneling server replies proxy acks to the end nodes to separate TCP connections. Thus, the problem that end nodes and tunneling servers retransmit the same TCP packet is solved.

Reference [18] proposed TCP for sensor networks in hilly and mountainous areas. Real-time performance is required for the data transfer of the sensor data. In addition, mobile communication in hilly and mountainous areas frequently causes packet loss and communication interruption compared to communication in urban areas. Therefore, the receipt of the latest data is greatly delayed after recovering from communication interruptions by TCP head-of-line blocking. Using the same background as our study, they proposed a retransmission-controlled TCP that suppresses the retransmission control of TCP for real-time communication. Even though the behavior of retransmission-controlled TCP is similar to that of UDP, control other than retransmission control works; therefore, unlike our proposed method, it affects the control of the communication to be passed.

SOCKS [19] is a technology that relays the communication of the transition layer protocol to pass through an FW. In SOCKS, a proxy server is installed at the boundary between the external and internal and networks, and both external and internal communication is accepted. Upon receiving a connection, the SOCKS server authenticates the end-user as necessary, generates a notification, and connects to the destination node. Application correspondence is necessary to use SOCKS; it cannot be used transparently. In addition, there is a problem of throughput degradation because the SOCKS server receives and retransmits; a packet once.

Thus, the existing tunneling protocols have the following problems: they cause control conflicts due to TCP over TCP, they use protocols and port numbers that may be blocked by FWs, and they cannot be used in NAT/NAPT environments. Therefore, it is insufficient for our purpose of FW and NAT/NAPT traversal.

3. Proposed Method

In previous studies, we proposed an encapsulation technique using pseudo-TCP to achieve FW traversal without losing the characteristics of the transport layer protocol. In this section, we outline the proposed method and describe the changes we made for our previous study.

3.1 Overview

The proposed method encapsulates packets created at the transport layer, such as UDP packets. Encapsulation is performed with a pseudo-TCP header in which port 443, which is used in HTTPS, is designated as the destination port. As a result, the FW recognizes a packet subjected to the processing of the proposed method as an HTTPS packet and allows it to pass through the FW. The proposed pseudo-TCP only performs a three-way handshake and connection management. In the proposed method, a pseudo-TCP header is inserted into the packet and transmitted. The packet with the inserted pseudo-TCP header passes through the FW and discards the pseudo-TCP header to return to its original form. This process enables communication with the original packet. To pass through stateful inspection FW, each field value of the inserted pseudo-TCP header must be set by emulating actual TCP communication. Therefore, a three-way handshake is emulated at the start of communication with the proposed method and periodic acknowledgments and sequence number management are performed during the communication.

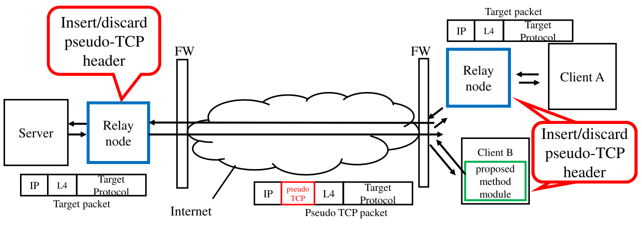


Fig. 1 System image of the proposed method.

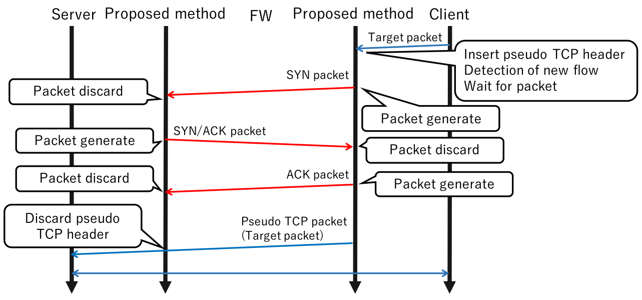


Fig. 2 An Emulated flow of a three-way handshake.

Figure 1 shows a system image of the proposed method. It is necessary to install a mechanism that implements the proposed method before and after the FW blocking communication. There are several options for the location of the proposed method.

For example, it can be embedded into end nodes such as clients and servers, into nodes on communication paths such as bridges and routers of each network, and into newly installed bridges and routers that implement the proposed method in each network. The case in which the proposed method is implemented outside the end-nodes has already been implemented and evaluated in our previous study. In this paper, we apply the proposed method to be end-nodes.

Figure 2 shows an emulated flow of a three-way handshake. When the proposed mechanism on the client-side receives the communication start packet of the target protocol, the proposed mechanism waits for the packet and sends/receives synchronize (SYN) packets, SYN/Acknowledge (ACK) packets, and ACK packets between the proposed mechanisms. These communications simulate a three-way handshake between the client and server using the IP address of the client and server. Then, a pseudo-TCP header is inserted into the waiting packet, which is then transmitted.

The packet to which the proposed method is applied is flagged in the reserved area of the pseudo-TCP header to distinguish it from normal HTTPS communication. Pseudo-TCP does not perform retransmission control or congestion control; therefore, double control does not occur, unlike in other existing technologies. In other words, the proposed method achieves FW traversal via encapsulation without losing the superiority of the target protocol.

3.2 Improvements with Respect to Our Previous Study

As described above, this paper implemented the proposed method module on the end-node. There are many middleboxes in actual networks, and there is a concern that middleboxes modify packets and may affect the communication of the proposed method. Therefore, we improved the proposed method to allow it to work properly even in an environment where NAPT is a typical middlebox. Accordingly, we changed the proposed method so that communication is possible even if NAPT, which is a typical middlebox, is included in the communication path.

The first improvement is the checksum of the L4 header of the encapsulated packet. When NAPT is applied, the checksum value of the L3 and L4 headers is recalculated with the conversion of the IP address and port number. However, in the case of a packet with an inserted pseudo-TCP header, the checksum of the pseudo-TCP header is recalculated when NAT is applied. Therefore, a checksum error occurs when receiving the encapsulated packet. As a countermeasure, we added a process to recalculate the L4 checksum of the encapsulated packet when the pseudo-TCP header is discarded.

The second improvement is support for port conversions by NAPT. NAPT converts the port number of the pseudo-TCP header when processing a packet with the pseudo-TCP header. Therefore, the port number in the L4 header of the packet received by the end node has not been rewritten and the end-node responds with the port number as the destination port number. However, the destination port number is not the value assumed by NAPT, and the proper port conversion is not performed. Accordingly, the proposed mechanism maintains a correspondence table between the port number of the pseudo-TCP header and the port number of the L4 layer protocol header of the encapsulated packet and sets an appropriate port number for the reply packet.

4. Implementation of the Proposed Method

To apply the proposed method to Linux, we implemented it as a loadable kernel module (LKM) using Netfilter. Netfilter is a framework for packet filtering and NAPT. In this section, we describe the processing of the proposed method with end-nodes.

4.1 Processing of LKM at the End-Nodes

Figure 3 shows an overview of the LKM process running on an end-node. The pseudo-TCP header insertion process is performed at the hook point POST_ROUTING through which all transmitted packets pass. The procedure for pseudo-TCP header insertion processing is shown below.

1) Determine whether the hooked packet is a packet of the protocol to which the proposed method has been applied. If it is a target packet, go to step 2. If it is not a target packet, go to step 5.
2) Identify the flow is from the four-tuple (i.e., the source IP address, destination IP address, source port number, and destination port number). For a new flow, a new flow management structure is defined. The flow management structure is a structure for managing the pseudo-TCP connection of the proposed method and includes information such as the four-tuple, sequence number, acknowledgment number, and state of the pseudo-TCP connection.
3) Perform pseudo-TCP header insertion processing. The pseudo-TCP header insertion process is set for each field by referring to the flow management structure. If the identification result in step 2 is a new flow, go to step 4. If the identification result in step 2 is an existing flow, go to step 5.
4) In the case of a new flow, wait for a packet with an inserted pseudo-TCP header inserted and start emulating a three-way handshake (client-side LKM only).
5) Return the packet to the hook point.

In our previous study, pseudo-TCP header insertion processing secured the storage area for the pseudo-TCP headers by shifting the IP header forward. However, in the end-node version, the hook point is different and, when the pseudo-TCP header is inserted, the free space in the socket buffer is less than 20 bytes. The end-node version secures the pseudo-TCP header storage area by shifting the L4 datagram to be encapsulated backward. Therefore, the end-node version of pseudo-TCP header insertion processing is expected to increase the overhead compared to t hat of our previous study.

The pseudo-TCP header discarding process is performed at the hook point PRE _ROUTING through which all received packets pass. The procedure for the pseudo-TCP header discard processing is as follows.

1) Determine whether the hooked packet is a packet with an inserted pseudo-TCP header. If it is a packet with an inserted pseudo-TCP header, go to step 2. If it is not a packet with an inserted pseudo-TCP header, go to step 5.
2) Identify the flow from the four-tuple.
3) Discard the pseudo-TCP header. The flow management

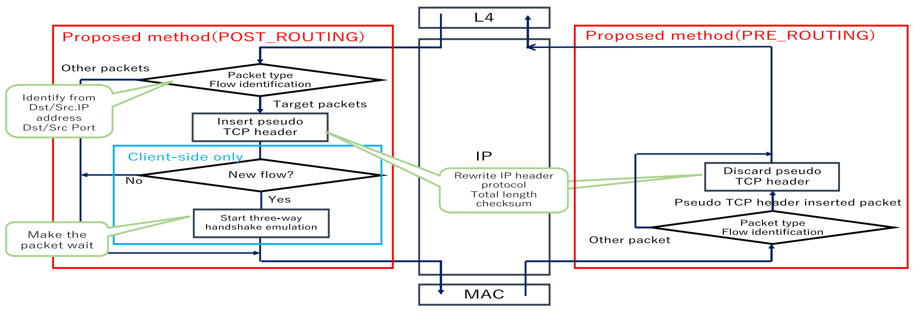


Fig. 3 Processing of LKM running on end-nodes.

structure is updated by referring to the field values of the inserted pseudo-TCP header.

4) Recalculate the checksum of the L4 header of the original packet.
5) Return the packet to the hook point.

The pseudo-TCP header discarding process is realized by shifting the IP header backward, both in the end-node version and in the version in our previous study. Therefore, the overhead of the discard processing is considered to be the same in both systems.

4.2 The Proposed Method Corresponding to NAPT

To use the proposed method in a real environment, it is necessary to consider the many middleboxes that exist in addition to the FW. There are some middleboxes that modify packets, and this may affect the proposed method. Therefore, measures were taken against NAPT, which is a typical middlebox.

NAPT is a mechanism that allows hosts in a private network to communicate with hosts on the Internet. When a host in the private network sends a packet to a host on the Internet, NAPT overwrites the source IP address and source port number to the global IP address and port number of the host on which NAPT is running. Simultaneously, the checksums field in the L4 and L3 headers are also updated, and the IP address and port number before and after the conversion are registered in the translation table. As a result, the host on the Internet appears to be communicating with the host where NAPT is running. When NAPT receives the response from the host on the Internet, it rewrites the L3 and L4 headers based on the translation table, and forwards the response to the host on the private network, which is the original communication source. This provides the hosts on the private network with reachability to the Internet. In order to use the proposed method in a NAPT-mediated environment, we need to consider the following two points.

The first point is the checksum value of the real L4 header of the target packet. NAPT also recalculates the L3 and L4 checksum values as the IP address and port number are converted. However, when NAPT is applied to a packet with an inserted pseudo-TCP header inserted, it is the checksum of the pseudo-TCP header that is recalculated and the checksum of the real L4 header is not recalculated. After that, the pseudo-TCP header is discarded and received by the end-node. However, because the checksum value is incorrect, packet reception at the end-node fails, the packet is discarded, and communication fails. As a countermeasure, the pseudo-TCP header is discarded and the checksum of the L4 header is recalculated after returning to the original packet.

The second is NAPT port conversion. When NAPT processes a packet with an inserted pseudo-TCP header, it converts the port number of the pseudo-TCP header. However, as with the checksum problem described above, the header of the encapsulated packet is not affected by NAPT. Therefore, the port number of the real L4 header cannot be rewritten and the end-node receives the target packet prior to NAPT port number conversion. However, because the destination port number is not the value that NAPT expects as a return packet, it is assumed that the port conversion could not be performed properly. To solve this problem, the port number of the pseudo-TCP header inserted by the proposed LKM method must be set to an appropriate value. In other words, LKM records the port number of the discarded pseudo-TCP header and stores the port number of the discarded pseudo-TCP header in the reply packet. This processing was implemented in both the end-node version and the relay-node version.

5. Evaluation

In this section, we describe the functional evaluation and performance evaluation of the proposed method. The proposed method targets not only protocols to be developed in the future, but also existing protocols such as QUIC, SCTP [20], DCCP [21], and RTP [22]. We adopted QUIC from them for the evaluation experiments. The reasons are as follows:

- QUIC is a general purpose protocol with the communication control mechanisms equivalent to TCP.
- There is software for both client and server that is close to the practical environment.

SCTP is supported by iPerf3, but there is no SCTP-based web server and client software. On the other hand, there are the QUIC-based web server, caddy, and the QUIC-based

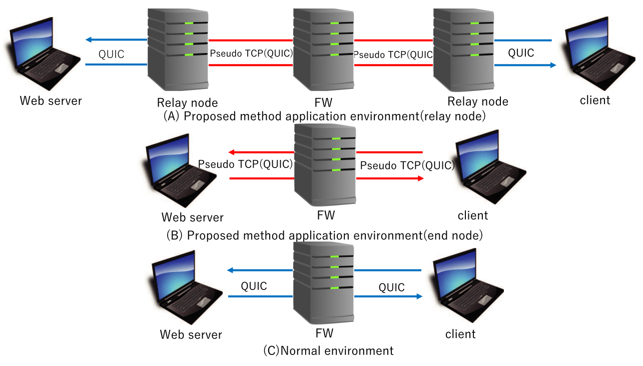


Fig. 4 Experimental enviroments.

```
*filter
:INPUT ACCEPT
:FORWARD DROP
:OUTPUT ACCEPT
-A FOWARD -p tcp ! --syn -m conntrack --ctstate NEW
-j DROP
-A FOWARD -m conntrack --ctstate INVALID -j DROP
-A FOWARD -p tcp --sport 80 -j ACCEPT
-A FOWARD -p tcp --dport 80 -j ACCEPT
-A FOWARD -p tcp --sport 443 -j ACCEPT
-A FOWARD -p tcp --dport 443 -j ACCEPT
-A FOWARD -p icmp -j ACCEPT
COMMIT
```

Fig. 5 Configuration of iptables.

web client, proto-quic [23][†]. Therefore, we excluded SCTP from our experiments. QUIC is a transport layer protocol that operates in UserLand and can handle multiple streams simultaneously. QUIC does not use TCP but communicates using UDP/443. However, there are many FWs that block UDP/443.

In the functional evaluation, we confirmed that QUIC communication was possible using the proposed method in an environment in which the FW blocks QUIC communication. In the performance evaluation, we measured the overhead caused by applying the proposed method. We confirmed the effect of the proposed method on the page download time and on the confliction to the encapsulated protocol's communication control.

5.1 Evaluation Environment

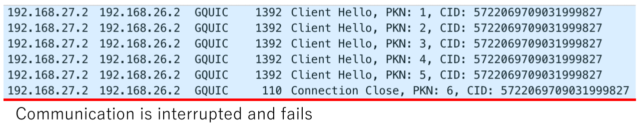
Figure 4 shows the evaluation environments. Figure 4 (A) shows the environment using the same relay-node version of the proposed method as in our previous study, Fig. 4 (B) shows the environment using the end-node version of the proposed method, and Fig. 4 (C) shows the environment without the proposed method. This is the general environment of the application.

The client used proto-quic, a QUIC test client. Caddy [25] was used on the web server. Caddy is a web server software package compatible with the Google version of QUIC. In the FW node, the FW that blocks QUIC communication was constructed using iptables. Figure 5 shows the iptables settings. For the settings in Fig. 5, everything except TCP/80, 443 and ICMP packets are blocked. In addition, using state matching via the conntrack module, a stateful inspection-type FW was built. In addition, NAPT was operated using iptables on the FW. The experimental scenario imitates general Web access that receives an HTML file on a web server (Caddy) using proto-quic. Table 1 shows the basic performance of the nodes used in the experiment.

[†]proto-quic is already out of development. Currently, it is being developed by The chromium project [24].

Table 1 Specification of experimental devices.

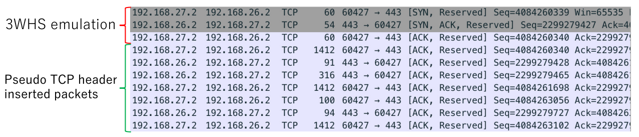
	End-nodes/FW	Client-side relay-node	Server-side relay-node
OS	Ubuntu16.04	CentOS7.2	CentOS7.4
Kernel	4.4.0	3.10.0	3.10.0
CPU	Core i3-4005U	Core i5-6600	Core i5-2320
Memory	4GB	8GB	8GB



```
192.168.27.2  192.168.26.2  GQUIC   1392 Client Hello, PKN: 1, CID: 5722069709031999827
192.168.27.2  192.168.26.2  GQUIC   1392 Client Hello, PKN: 2, CID: 5722069709031999827
192.168.27.2  192.168.26.2  GQUIC   1392 Client Hello, PKN: 3, CID: 5722069709031999827
192.168.27.2  192.168.26.2  GQUIC   1392 Client Hello, PKN: 4, CID: 5722069709031999827
192.168.27.2  192.168.26.2  GQUIC   1392 Client Hello, PKN: 5, CID: 5722069709031999827
192.168.27.2  192.168.26.2  GQUIC    110 Connection Close, PKN: 6, CID: 5722069709031999827
```

Communication is interrupted and fails

Fig. 6 Communication log in a normal environment (Fig. 4 (C)).



```
192.168.27.2  192.168.26.2  TCP    60  60427 → 443  [SYN, Reserved] Seq=4084260339 Win=65535
192.168.26.2  192.168.27.2  TCP    54  443 → 60427  [SYN, ACK, Reserved] Seq=2299279427 Ack=40
192.168.27.2  192.168.26.2  TCP    60  60427 → 443  [ACK, Reserved] Seq=4084260340 Ack=229927
192.168.27.2  192.168.26.2  TCP  1412  60427 → 443  [ACK, Reserved] Seq=4084260340 Ack=229927
192.168.26.2  192.168.27.2  TCP    91  443 → 60427  [ACK, Reserved] Seq=2299279428 Ack=408426
192.168.26.2  192.168.27.2  TCP   316  443 → 60427  [ACK, Reserved] Seq=2299279465 Ack=408426
192.168.27.2  192.168.26.2  TCP  1412  60427 → 443  [ACK, Reserved] Seq=4084261698 Ack=229927
192.168.27.2  192.168.26.2  TCP   100  60427 → 443  [ACK, Reserved] Seq=4084263056 Ack=229927
192.168.26.2  192.168.27.2  TCP    94  443 → 60427  [ACK, Reserved] Seq=2299279727 Ack=408426
192.168.27.2  192.168.26.2  TCP  1412  60427 → 443  [ACK, Reserved] Seq=4084263102 Ack=229927
```

Fig. 7 Communication log application environment of the end-node version of the proposed method (Fig. 4 (B)).

5.2 Functional Evaluation

In this evaluation, it was confirmed that the proposed method can pass through an FW blocking QUIC communication.

Figure 6 shows the packet capture log obtained by the NIC on the client-side of the FW in the environment of Fig. 4 (C). Because the UDP/443 port is not open, the packet is blocked by the FW and communication fails.

Figure 7 is a packet capture log obtained by the NIC on the client-side of the FW when the end-node version of the proposed method is applied. The log is shown in Fig. 7 is for after the proposed end-node method has been applied, that is, the sent packet is logged after the pseudo-TCP header insertion process and the received packet is logged before the pseudo-TCP header discard process. Therefore, the QUIC packet is disguised as a TCP packet using port number 443. The first three packets are the three-way handshake emulation packets, and the subsequent packets are QUIC packets with pseudo-TCP header insertion processing. From the Fig. 7, it can be confirmed that the bidirectional communication passed through the FW. We confirmed that QUIC communication was possible when applying the proposed method.

Table 2 Processing time per packet of the proposed method (unit: ns).

	End-node		Relay-node	
	Insertion	Discard	Insertion	Discard
Applying the proposed method	1405	1124	462	672
Without the proposed method	914	646	335	310

5.3 Evaluation of the Processing Overhead Per Packet

We measured the overhead of inserting and discarding pseudo-TCP headers when using the proposed method. SystemTap was used for the measurement, and the overhead was calculated by acquiring the time before and after the callback function of the proposed method was called. In the overhead measurement of the relay-node insertion/discard processing and the end-node discard processing, the time from when the ip_rcv function is called to when the ip_rcv_finish function is called was measured. The end-node insertion processing overhead was measured from the time when the ip_output function was called to the time when the ip_finish_output function was called. SystemTap was operated on the client for the end-node version and on the client-side relay node for the relay-node version.

Table 2 shows the average processing time per packet obtained by repeating the QUIC communication to download a 1 MB file three times. Because only the processing overhead of the proposed LKM method was measured, the overhead of the relay-node version did not include the overhead of the packet relay processing. The end-node version insert packet was a QUIC acknowledgment packet, and the packet size was small. As described in Sect. 4.1, end-node version insert processing reserves the area of the pseudo-TCP header by copying the L4 datagram backward. Therefore, the overhead depends on the packet size. The overhead is small but can not negligible. Modification of the buffer allocation scheme is effective in decrease the overhead. Since the buffer allocated for the packet does not have enough space to insert pseudo-TCP header in front of the packet, a packet copy is required for the insertion. The effect of the copy increases the overhead with the packet size. Therefore, the buffer allocation taking the insertion account will mitigate the overhead related to the packet copy.

5.4 Measuring the Impact on the Page Download Time

We measured the effect of the overhead of the proposed method on the QUIC page download time.

5.4.1 Measurement Method

In the experiment, the client downloaded an HTML file from the server and measured the time required for the download. The experiment was performed with three different HTML file sizes (1 MB, 5 MB, and 10 MB). We used the tc command in the FW to simulate the actual Internet and made the measurement in five different delay environments (0 ms, 20 ms, 50 ms, 70 ms, 100 ms). The experimental pattern was 45 patterns with combinations of file sizes and delays, and we tried each pattern 100 times. The page download time was calculated using the timestamp value of the packet capture log acquired on the client. The page download time indicates the time from the first captured QUIC packet to the last QUIC packet received by the client in a series of communications. In this measurement, QUIC was used in three environments: Fig. 4 (A) with the relay-node version of the proposed method, Fig. 4 (B) with the end-node version of the proposed method, and Fig. 4 (C) without the proposed method. We initiated the communication and compared the web page download times.

Table 3 Page download time measurement result (unit: ms).

File size		Delay				
		0ms	20ms	50ms	70ms	100ms
1MB	Normal environment	107	205	415	552	767
	Proposed method (relay node)	115	224	470	630	869
	Proposed method (end node)	106	227	467	627	864
5MB	Normal environment	362	549	846	902	1204
	Proposed method (relay node)	366	565	849	968	1305
	Proposed method (end node)	363	589	905	1011	1314
10MB	Normal environment	672	867	1149	1258	1669
	Proposed method (relay node)	680	883	1204	1328	1792
	Proposed method (end node)	677	908	1255	1385	1809

5.4.2 Measurement Results and Discussion

Table 3 shows the measurement results. The normal environment is the page download time in the environment of Fig. 4 (C), and the proposed method (relay) is the page download time in the environment of Fig. 4 (A), and the proposed method (end) is the page download time in the environment of Fig. 4 (B). Table 4 shows the differences between the proposed method application environments and the normal environment. Table 4 proves that the effect of the proposed method increases as the network delay increases. This is caused by the three-way handshake emulation of the proposed method. Using the three-way handshake emulation, communication is started after the SYN/ACK packet is received after sending the SYN packet. Therefore, the start of communication is delayed by the round-trip time (RTT) between the nodes when using the proposed method. Therefore, in an environment in which the proposed method is applied, the page download time for 1 RTT increases compared to the normal environment. Accordingly, the three-way handshake emulation of the proposed method was excluded, that is, the value obtained by subtracting 1 RTT from the measurement results when both proposed methods were applied (Table 4).

In an environment with a file size of 1 MB, the measurement results of both proposed methods and the normal environment are almost identical. Therefore, in the envi-

Table 4 Differences between applications of the proposed method and the normal environment (unit: ms).

File size		Delay				
		0ms	20ms	50ms	70ms	100ms
1MB	Proposed method(relay)	8	19	55	78	102
	Proposed method(end)	-1	22	52	75	97
5MB	Proposed method(relay)	4	16	3	66	101
	Proposed method(end)	1	40	59	109	110
10MB	Proposed method(relay)	8	16	55	70	123
	Proposed method(end)	5	41	106	127	140

ronment with a file size of 1MB, the effect of the proposed method is due to the three-way handshake emulation, and the effect of the pseudo-TCP header insertion and discarding process is hardly seen at all.

In environments with file sizes of 5 MB and 10 MB, the page download time increases only when the end-node version of the proposed method is applied. The effect is particularly noticeable when the delay is 20 ms, 50 ms, or 70 ms. This is thought to be because the number of packets per unit time increased compared to when the file size was 1 MB, and the effect of the pseudo-TCP header insertion/discard processing appeared. In addition, as described in Sect. 4.1, the overhead of pseudo-TCP header insertion processing differs between the relay-node version and the end-node version. In the relay-node version, a memory copy of 20 bytes occurs for each packet. Conversely the other hand, in the end-node version, a memory copy of up to 1350 bytes occurs for each packet. Therefore, the end-node version of the proposed method is more likely to affect the page download time.

From this evaluation, it is clear that the overhead of the proposed method is primarily caused by the emulation of a three-way handshake. As a minimum overhead, there is a delay of 1 RTT between the nodes when the proposed method is operated, and additional overhead is caused by the insertion/discarding of the pseudo-TCP header. Therefore, it is thought that the impact of the proposed method will be noticeable on networks with large RTT. However, three-way handshake emulation is performed only once when target communication with the proposed method is started. Therefore, it is expected to have a limited effect because the increase in the communication time caused by a three-way handshake is inversely proportional to the total QUIC communication time. Actually, in this evaluation applied to QUIC, it is approximately 60 ms at its maximum and it is approximately 4% of the entire communication and is, therefore, considered to be a small influence.

5.5 Performance Comparison of TCP, QUIC, and the Proposed Method Applied to QUIC

In this section, we compare the proposed method applied to QUIC, QUIC, and TCP (HTTPS) and confirm the effectiveness of the proposed method.

5.5.1 Measuring Method

The evaluation environments are shown in Fig. 4, as in the previous evaluation. Curl was used for the TCP (HTTPS) communication clients. Similar to Sect. 5.4, we used three different sizes of HTML files and simulated the actual Internet using the tc command. There were three delay parameters: 0 ms, 20 ms, and 50 ms. There were eight packet loss rates: 0.1%, 0.3%, 0.5%, 0.7%, 0.9%, 1.1%, 1.3%, and 1.5%. The experimental patterns consisted of 288 patterns combining the protocols (TCP, QUIC, end-node version of the proposed method applied to QUIC, relay-node version of the proposed method applied to QUIC), file size, delay, and packet loss, and each pattern was tried 100 times. The measurement interval was the same as in Sect. 5.4.

5.5.2 Measurement Results and Discussion

Figure 8 shows the measurement results. From the measurement results, it was confirmed that QUIC using the proposed method showed communication performances close to those of the normal QUIC even in an environment with packet loss and delay. Basically, QUIC using the proposed method had a better communication performance than TCP. As an exception, QUIC using both proposed method in the results of that RTT is 0ms, bandwidth is 5MB or 10MB had a lower performance than TCP. However, the performance of QUIC without the proposed method was lower than that of TCP. In the literature [26], it is mentioned that using QUIC in networks with a wide bandwidth (over 100 Mbps), low latency (several ms), and low packet loss rates may result in a lower performance than TCP. This environment is not considered to be a general Internet state, and it is assumed in the paper that such a state is caused by a client-side scheduler. The conditions of wide bandwidth and low delay match the cases of that QUIC and the proposed methods defeated TCP. Except in these two environments, QUIC outperforms TCP and the proposed method applied to QUIC outperforms TCP even though the communication performance is slightly lower than that of QUIC alone. This result indicates that the application of the proposed method to QUIC is effective in the network of this evaluation environment. The proposed method applied to QUIC has slightly lower communication performance compared to normal QUIC, however, it keeps the advantage of QUIC against TCP. This result indicates that the application of the proposed method to QUIC is effective in the network of this evaluation environment.

In this evaluation, the 0-RTT handshake of QUIC was disabled. Therefore, in QUIC in a real environment, there should be a further difference with TCP when communicating with a server that has communicated once. However, when applying the proposed method, even if QUIC can start communication with the 0-RTT handshake, the three-way handshake emulation of the proposed method causes a delay of 1 RTT. In addition, because the proposed method iden-

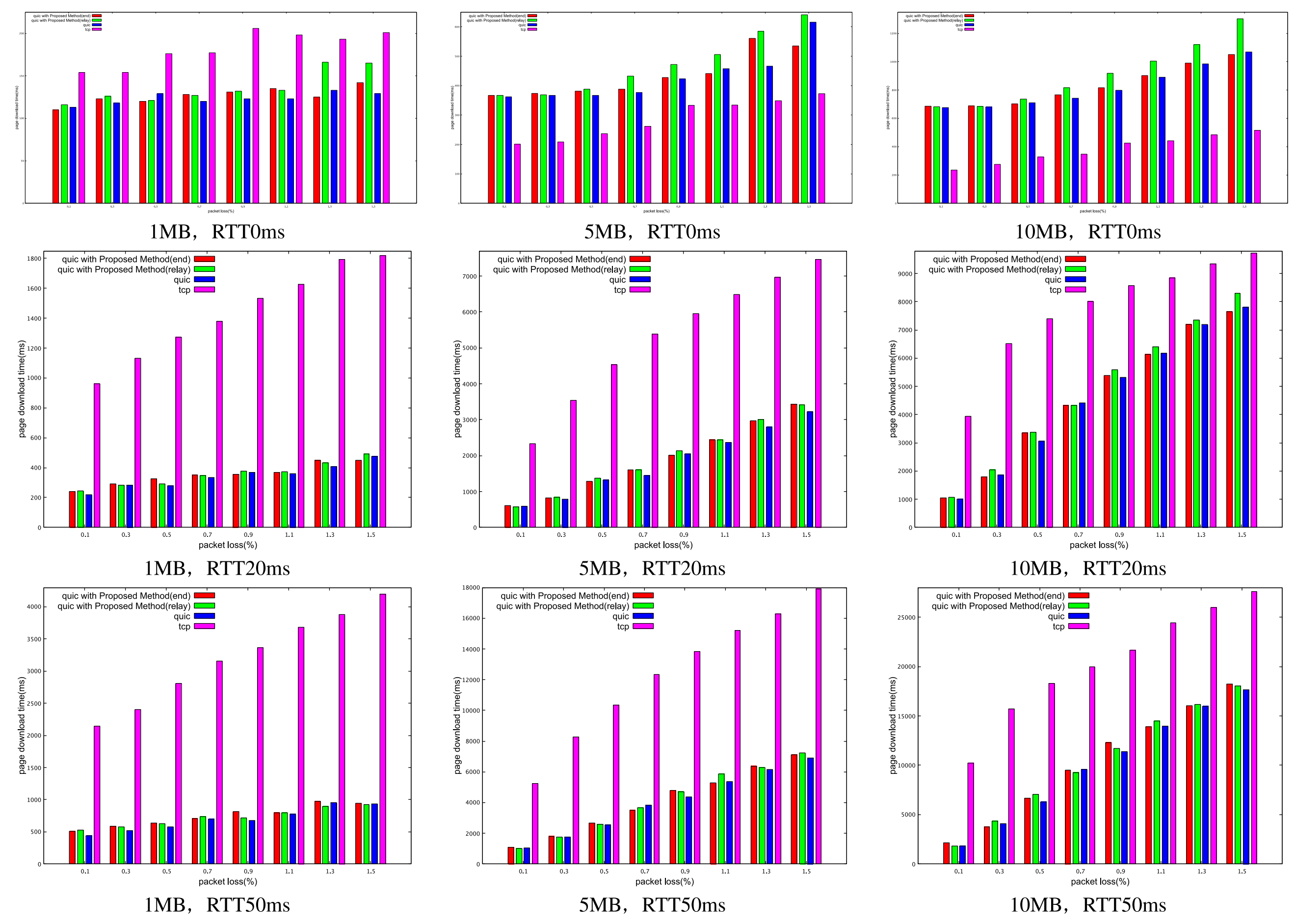


Fig. 8 Performance Evaluation of Proposed Method QUIC/Normal QUIC/TCP (HTTPS)

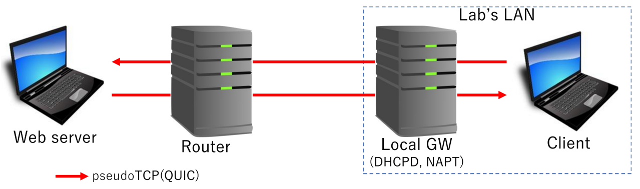


Fig. 9 Practical environment.

Table 5 Machine spec of the practical experiment.

	Client	Server
OS	Ubuntu16.04	Ubuntu16.04
Kernel	4.4.0	4.4.0
CPU	Core i3-4005U	Xeon x3460
Memory	4GB	8GB

tifies flows in four-tuple, there remains the problem that a delay of 1 RTT occurs due to the three-way handshake emulation of the proposed method even during QUIC connection migration.

5.6 Evaluation in Practical Environment

To verify the practicability of the proposed method, we evaluated the proposed method in the practical environment including the Internet. We used the end-node type proposed method for the verification.

5.6.1 Environment

The environment is illustrated in Fig. 9. The client is inside of our laboratory's LAN. The web server belongs to the outside network. Therefore, their communication passes through the Internet. The spec of the client and the server is as Table 5. Same as the experiments mentioned above, the client used proto-quic and curl, the server used Caddy.

5.6.2 Experimetal Results

We verified the practicability of the proposed method by the comparison of page download time. We measured the page download time of QUIC with the proposed method, normal TCP, and normal QUIC, respectively. The time used for comparison is mean time of the 100 times experimental results. In addition, we experimented with different size HTML file 1MB, 5MB, 10MB. To measure the page download time, we ran the WireShark on the cliant, and calculated the time based on time stamp of captured packets.

Figure 10 illustrates the experimental results. The dif-

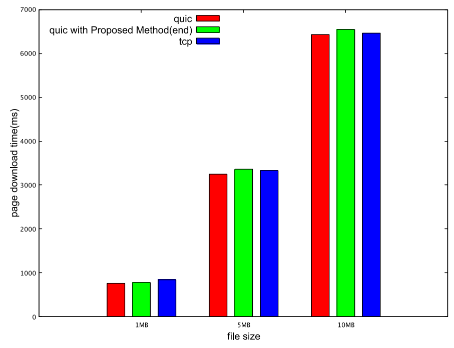


Fig. 10 Experimental results in the practical environment.

ference among them is a very little. However, in the cases of 5MB and 10MB, the page download time of QUIC with the proposed method is the largest in the three. The network performance between the client and the server was that RTT was about 15ms, and the packet loss rate was 0%. Namely, the high network performance lead to almost the same tendency as in Fig. 8. Therefore, when network condition is good, the proposed method increase communication time a little, although, the proposed method will give less affect in bad network conditions.

6. Conclusions

In this paper, the proposed method was implemented in the end-node and was modified to work in an environment where NAPT exists. As in our previous study, we evaluated QUIC and confirmed that it was able to pass through a stateful inspection FW that confirmed the TCP header in detail. The performance evaluation indicated that the proposed method affected the page download time. There is, at minimum, a three-way handshake emulation delay, in addition, to be pseudo-TCP header insertion and discard process delay. However, this delay is small from the viewpoint of the entire communication, and it was confirmed that its influence was small. In addition, QUIC using the proposed method basically has a higher communication performance than TCP (HTTPS) and it is considered that the proposed method is effective for QUIC.

In the current implementation, the application of the proposed method obstructs characteristics of QUIC such as 0-RTT handshake and connection migration. To solve this problem, TCP Fast Open emulation and a flow identification method using parameters other than a four-tuple are the future work.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Number 18K18045.

References

[1] K. Taga, J. Zheng, K. Mouri, S. Saito, and E. Takimoto, “Firewall traversal method by inserting pseudo tcp header into quic,” IEICE Technical Report, vol.305, pp.87–92, 2018.
[2] K. Taga, J. Zheng, K. Mouri, S. Saito, and E. Takimoto, “Firewall traversal method by inserting pseudo tcp header into quic,” Lecture Notes in Engineering and Computer Science, vol.2239, pp.216–221, 2019.
[3] J. Iyengar and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport,” RFC 9000, May 2021.
[4] O. Honda, H. Ohsaki, M. Imase, M. Ishizuka, and J. Murayama, “Understanding tcp over tcp: Effects of tcp tunneling on end-to-endthroughput and latency,” IEICE Technical Report, vol.104, no.438, pp.79–84, 2004.
[5] O. Titz, “Why tcp over tcp is a bad idea.” http://sites.inka.de/sites/big-red/devel/tcp-tcp.html, accessed July 23, 2021.
[6] D. Nobori, “Virtual ethernet system and tunneling communication with softether,” Proc. of Programming Symposium, pp.147–158, IPSJ, 2004.
[7] T. Li, D. Farinacci, S.P. Hanks, D. Meyer, and P.S. Traina, “Generic Routing Encapsulation (GRE),” RFC 2784, March 2000.
[8] K. Seo and S. Kent, “Security Architecture for the Internet Protocol,” RFC 4301, Dec. 2005.
[9] L. Yong, E. Crabbe, X. Xu, and T. Herbert, “GRE-in-UDP Encapsulation,” RFC 8086, March 2017.
[10] V. Volpe, M. Stenberg, B. Swander, L. DiBurro, and A. Huttunen, “UDP Encapsulation of IPsec ESP Packets,” RFC 3948, Jan. 2005.
[11] G. Zorn, G.S. Pall, and K. Hamzeh, “Point-to-Point Tunneling Protocol (PPTP),” RFC 2637, July 1999.
[12] M. Littlewood, A. Valencia, T. Kolar, T. Kolar, and T. Kolar, “Cisco Layer Two Forwarding (Protocol) “L2F”,” RFC 2341, May 1998.
[13] M. Townsley, I. Goyret, and J. Lau, “Layer Two Tunneling Protocol - Version 3 (L2TPv3),” RFC 3931, March 2005.
[14] P. Garg and Y.S. Wang, “NVGRE: Network Virtualization Using Generic Routing Encapsulation,” RFC 7637, Sept. 2015.
[15] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, “Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks,” RFC 7348, Aug. 2014.
[16] B. Davie and J. Gross, “A Stateless Transport Tunneling Protocol for Network Virtualization (STT),” Internet-Draft draft-davie-stt-08, Internet Engineering Task Force, April 2016. Work in Progress.
[17] S. Ogawa, K. Yamazaki, R. Kawashima, and H. Matsuo, “T3: Tcp-based high-performance and congestion-aware tunneling protocol for cloud networking,” Proc. on ICCCRI, pp.64–70, IEEE, 2016.
[18] S. Yokoyama, H. Yamamoto, and K. Yamazaki, “The evaluation of communication characteristic of cellular network and implementation and evaluation of retransmission-controlled tcp,” IEICE Trans. Inf. & Syst. (Japanese edition), vol.95, no.5, pp.1133–1141, May 2012.
[19] M.D. Leech, “SOCKS Protocol Version 5,” RFC 1928, March 1996.
[20] R.R. Stewart, “Stream Control Transmission Protocol,” RFC 4960, Sept. 2007.
[21] S. Floyd, M.J. Handley, and E. Kohler, “Datagram Congestion Control Protocol (DCCP),” RFC 4340, March 2006.
[22] H. Schulzrinne, S.L. Casner, R. Frederick, and V. Jacobson, “RTP: A Transport Protocol for Real-Time Applications,” RFC 3550, July 2003.
[23] “proto-quic.” https://github.com/google/proto-quic (Obsoleted).
[24] “The chromium project.” https://www.chromium.org/, accessed July 23, 2019.
[25] “caddy.” https://caddyserver.com accessed Feb. 15, 2021.
[26] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.T. Chang, and Z. Shi, “The quic transport protocol: Design and internet-scale deployment,” Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM ’17, New York, NY, USA, pp.183–196, ACM, 2017.

Keigo Taga received the B.E., the M.E. degree in computer science from Ritsumeikan University in 2018 and 2020, respectively. He is with NDR, Co., Ltd.. His research interests include computer network and IoT technology.

Junjun Zheng received the B.S.E. degree in engineering from Fujian Normal University, Fuzhou, China, in 2010, and the M.S. and D.Eng. degrees in engineering from Hiroshima University, Higashihiroshima, Japan, in 2013 and 2016, respectively. In 2016 and 2017, he was a Visiting Researcher with the Department of Information Engineering, Graduate School of Engineering, Hiroshima University. Since 2018, he has been an Assistant Professor with the Department of Information Science and Engineering, Ritsumeikan University, Japan. His research interests include performance evaluation and dependable computing. Dr. Zheng is a member of the Operations Research Society of Japan, the Reliability Engineering Association of Japan, the Institute of Electrical, Information and Communication Engineers, and the Institute of Electrical and Electronics Engineers.

Koichi Mouri received the B.E., the M.E. and the Ph.D degree in computer science from Ritsumeikan University in 1994, 1996 and 2000, respectively. He is a professor in College of Information Science and Engineering, Ritsumeikan University. His research interests include operating systems, computer security and computer network. He is a member of the ACM, IEEE Computer Society, and Information Processing Society of Japan.

Shoichi Saito received the B.E., the M.E.received his B.S. and M.E. degrees in engineering from Ritsumeikan University in 1993 and 1995, and became a research associate in the Department of Computer and Communication Sciences, Wakayama University in 1998. He received his Dr. Eng. degree in 2000. He was an assistant professor from 2003 and an associate professor from 2005. He was an associate professor in Nagoya Institute of Technology from 2006 and has been a professor from 2016. His research interests are operating systems, security, and the Internet. He is a member of ACM, IEEE CS, and IPSJ.

Eiji Takimoto received the B.E., the M.E. and the Ph.D degree in computer science from Ritsumeikan University in 1998, 2000, and 2015, respectively.He was an assistant professor in Ritsumeikan University from 2016. He is a lecturer in Hiroshima Institute of Technology from 2020. His research interests include computer security, computer network. He is a member of Information Processing Society of Japan.