The Case for Protocol Plugins

Quentin De Coninck*
UCLouvain, Belgium
quentin.deconinck@uclouvain.be

Olivier Bonaventure
UCLouvain, Belgium
olivier.bonaventure@uclouvain.be

ABSTRACT

Transport protocols such as TCP, SCTP or QUIC are supposedly extensible thanks to their flexible packet formats. However, implementations also need to be modified to support such extensions and changing those extensions remains difficult. Furthermore, it is difficult for an application to finely tune the underlying protocol to its needs.

Our proposed *protocol plugins* address these two needs. A *protocol plugin* is a small executable code which can be dynamically plugged inside an implementation on a per-connection basis. We first propose a methodology to modify an existing implementation to support protocol plugins. We apply this methodology to two different QUIC implementations written in C and Go. We then demonstrate how servers can extend client stacks with protocol plugins that implement Tail Loss Probe, Explicit Congestion Notification, pacing rate and different acknowledgement strategies. We then discuss how protocols should leverage *protocol plugins*.

1 INTRODUCTION

Extensibility is an important feature that is found in successful Internet protocols, including TCP, HTTP, TLS, ... To be expandable, a protocol must use an extensible syntax for its messages. TCP [42] relies on TCP options. These options are structured as variable-length Type-Length-Value fields which can be placed in the extended segment header. Over the years, a variety of TCP options have been defined [13]. However, the limited size of the TCP extended header is now a severe constraint on the extensibility of TCP. SCTP [48] solved this problem by encoding each packet as a fixed header followed by a series of chunks. The QUIC protocol [34], went one step further by introducing frames and 32 bits version numbers.

Given their extensible syntax, it should be simple to extend transport protocols. This is true when considering a single implementation. However, deploying a TCP extension is much more challenging [30]. It requires coordination among implementers who need to agree on the specification of the modification, implement it correctly and then ship the modified code. This process takes several years or more. It took almost a decade to fully deploy TCP Selective Acknowledgements, Window Scale and Timestamps [22]. The deployment of TCP Fast Open [8] is still ongoing [49]. In the early days, support for a new extension was required on both clients *and* servers. Nowadays, middleboxes such as firewalls or load balancers also need to be upgraded to enable the deployment of new protocol extensions [30, 47]. This has resulted in a growing ossification of the transport layer.

QUIC addresses this ossification in two ways [34]. First, QUIC encrypts and authenticates almost all packets and most of its packet headers. This prevents middleboxes from interfering with the protocol itself. Second, QUIC is implemented above UDP as a regular application or a library. This simplifies the deployment of new versions of the protocol and contrasts with TCP and SCTP that are usually implemented inside the operating system and require OS upgrades to deploy new features. Measurements indicate that Google changes the QUIC version used by Chrome several times per year [46]. However, only the (few large) organisations that implement and deploy both servers and clients can extend their QUIC stack so easily.

Another problem with current transport protocols is that it is very difficult for an application to tune the underlying implementation to its needs. A chat application does not need the same features as a video streaming application or a bulk transfer. TCP stacks [15] support a small number of socket options which can be applied on a per-connection basis (e.g., enabling the Nagle algorithm, configuring keep-alives or the windows) and system-wide configuration parameters (e.g., activation of TCP extensions, maximum windows, or congestion control mechanisms).

In this paper, we argue for truly extensible transport protocols. We believe that implementations should be easily customisable by both peers on a per-connection basis. In Sect. 2, we revisit protocol implementations by introducing *anchors* specific to *protocol operations* in order to *inject plugin code* run inside *virtual machines*. Section 3 explores the feasibility and the potential overhead of this design by instrumenting two different QUIC implementations. Section 4 concludes this paper by discussing open questions raised by our proposed design.

2 PROTOCOL PLUGINS

In this section, we advocate for a design that enables protocol implementations to be much more customisable by supporting the insertion of *protocol plugins*. A *protocol plugin* is a small block of executable code which can be dynamically inserted in an implementation to modify its operation on a per-connection basis. For this, we first need to identify the core operations in the protocol. Implementations then require anchors where plugins can be inserted. Finally, plugins need an execution environment in which they can be run.

Pluggable Operations. To support *protocol plugins* we first need to identify and instrument the main operations of an

*FNRS Research Fellow

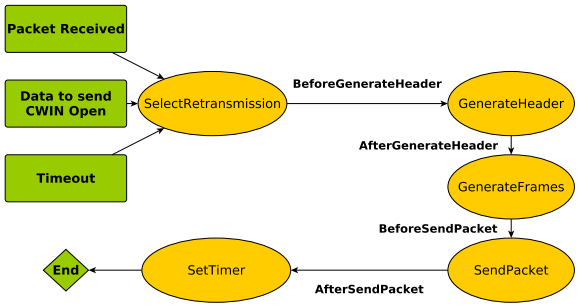

Figure 1: Simplified description of the sending process of QUIC.

implementation. From the protocol specification and its implementation, we identify high-level operations. These perform actions that produce a given outcome. We call these high-level procedures *pluggable operations*. Each *pluggable operation* has its own specification. A first example of *pluggable operation* is the congestion control algorithm. It reacts to specific events (e.g., packet loss, packet marking, rtt measurements, ...) and updates the congestion window as well as its own state. Various congestion control schemes [6, 27] have been implemented as modules in the Linux kernel. Our proposal matches calls for increased flexibility with congestion control schemes [37]. Another example is the computation of the retransmission timer. Various methods have been proposed to compute this timer [40, 41, 50]. Other operations are also made pluggable, such as packet sending, packet reception or the generation of protocol messages.

As a more detailed example, let us consider how a QUIC implementation sends frames. The QUIC specification [33] does not yet include a Finite State Machine (FSM) with explicit states and transitions, but it is possible to infer them from the specification and the available implementations. Figure 1 shows a simplified overview of the operations performed when sending a QUIC packet. This process can be triggered by the reception of a (processed) packet, a request to transmit application data when congestion window is open or the expiration of a timer. At this point, a QUIC implementation first checks whether an outstanding packet needs to be retransmitted. Then, the content of the packet, i.e., the public header and the frames, is generated. These frames can either be new ones or retransmissions, depending on the outcome of the first operation. Then, the packet is protected and sent over the network. Finally, since QUIC ensures reliable delivery, the sending host sets the retransmission timer.

In this example, we identify five *pluggable operations*: *SelectRetransmission*, *GenerateHeader*, *GenerateFrames*, *SendPacket* and *SetTimer*. Implementations provide a default behaviour for each *pluggable operation*. This behaviour can be bypassed by the injecting of *plugins*. For instance, an extension can compute a different value for the retransmission timer by inserting a plugin for the *SetTimer* operation. In addition, *transitions* between the operations are also made pluggable. By default, these transitions are no-ops.

Inserting Plugins. We use the name *plugin anchors* to identify the locations within a protocol implementation where a *protocol plugin* can be inserted. There are two types of *plugin anchors*. The first type corresponds to a *pluggable operation*. In this case, the plugin can replace the default behaviour associated to the pluggable operation. For instance, an extension can implement the Nagle algorithm [36] by preventing *SendPacket* to send a too short packet, and delaying it using *SetTimer* and reconsidering it in *SelectRetransmission*. The other type corresponds to *pluggable transitions*. These allow extending the protocol by adding processing which is different from existing pluggable operations. For example, an extension might want to insert a plugin for the *AfterSendPacket* transition to compute the current packet transmission rate and adapt the behaviour of the sender. We associate a unique identifier to each *plugin anchor* to enable the insertion of a protocol plugin at this specific location in the implementation. We also associate with this identifier the input parameters and the output that are expected from each plugin. To achieve plugin insertion, there are several ways to implement *plugin anchors* [10, 26, 38].

Running Plugins. The last point that is required to execute protocol plugins is an execution environment. A simple approach would be to plug native code directly in the running implementation. This would efficiently utilise the available hardware, but would force developers to create a plugin for each specific implementation and hardware platform. Furthermore, native code has potentially full access to the protocol implementation, which creates obvious safety concerns. A better approach is to execute the *protocol plugins* inside a virtual machine. Various virtual machines have been proposed for different purposes [3, 17, 23, 28, 35, 51]. We have three main requirements for this virtual machine. Firstly, it must provide isolation between the protocol plugins and protocol implementation to ensure that a malicious plugin will not be able to break the protocol implementation or interfere with other connections. Secondly, it should be possible to execute it inside protocol implementations written in different programming langages. Thirdly, it must efficiently run the *protocol plugins*.

To meet these three requirements, we opt for the eBPF virtual machine [17] for similar reasons as Amit et al. [1], even if the context is different. eBPF is a modern variant of the BPF virtual machine that was introduced to support flexible packet filters [3]. The eBPF virtual machine is included in the Linux kernel since 2014 where it is used to support services like `seccomp` [14], tracing and performance monitoring [25] or some TCP extensions [5]. eBPF prevents several safety problems by ensuring memory isolation and there is a limit on the number of instructions that the eBPF bytecode can execute (eBPF is not Turing-complete and does not support infinite loops). Furthermore, it is possible to ensure that some eBPF bytecode will not access external memory or loop [52].

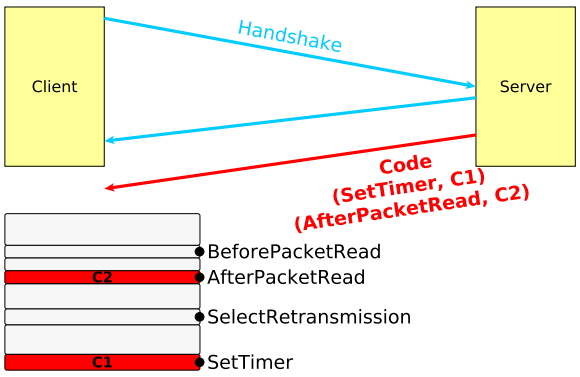

Figure 2: A server pushing code plugins to modify the client behaviour over a connection.

The eBPF implementation in the Linux kernel includes such a verifier. It is also possible to attach an eBPF virtual machine to a user-space application [32]. The absence of infinite loops in the eBPF virtual machine is beneficial from a safety viewpoint.

However, it restricts the operations that *protocol plugins* can execute since they cannot contain loops without clear variant. Furthermore, plugins also need to access the connection context and possibly maintain their own state. We solve these problems by introducing a dedicated API. First, it allows plugins to access and modify the current connection context provided by the implementation. Second, it provides a per-context memory area opaque to the implementation that can be used by plugins to store state between their executions. Third, it exposes helper functions implemented by the implementation to plugins. These helper functions provide basic services that are required by different plugins and would be difficult to implement entirely in eBPF. For example, we provide helper functions to allocate memory on the shared memory between the virtual machine and the implementation, set/get a socket option and read/write on a socket since the eBPF code cannot execute system calls.

2.1 How to Dynamically Attach a Plugin ?

The last element of our architecture is how *protocol plugins* can be dynamically attached to a running implementation. There are two possible approaches. A first possibility is to let an application inject a *protocol plugin* to extend the underlying implementation. This could be considered as a generalisation of the socket options that are supported by TCP implementations. For example, an interactive application could inject support for Tail Loss Probe [16] to tune TCP's retransmission strategy to its needs. An extension of TCP-BPF [5] could provide such a feature.

However, this is not the main use case for *protocol plugins*. The main benefits of our proposal will be realised when servers will be able to push *protocol plugins* on the clients after the connection handshake. Given the restricted size of the TCP options and the risk of middlebox interferences [30], this cannot be applied to TCP. However, this vision can be realised with a protocol such as QUIC [34]. Figure 2 illustrates our proposal. QUIC includes a secure handshake, encrypts and authenticates all the exchanged packets. After the secure handshake, the server can easily send *protocol plugins* as eBPF bytecode with their identifier to the client over a dedicated stream to modify the client implementation for this specific connection. The solution is symmetric and the client could also inject eBPF bytecode. It is likely that a similar approach would be applicable for other protocols where a connection starts with a secure handshake including TLS, SCTP over DTLS, of HTTP/2 over TLS.

3 A QUIC PROTOTYPE

To demonstrate the feasibility and the benefits of *protocol plugins*, we develop a series of plugins for two different implementations of QUIC [2]. The choice of QUIC is motivated by its clean and flexible design which makes it easy to define new frames. Furthermore, as QUIC packets are both encrypted and authenticated, middleboxes cannot interfere with the transmission of plugin code or our proposed extensions.

From a systems viewpoint, it should be possible to support *protocol plugins* on different implementations written in different programming languages. We first use `picoquic` [31], one of the C implementations being developed within the IETF to support the QUIC standardisation [33]. Our second implementation is `mp-quic` [11]. This is an extension of `quic-go`, a Go implementation of Google's variant of QUIC. This implementation supports Multipath QUIC [12].

To execute the *protocol plugins*, we link a user-space eBPF virtual machine written in C [32] to each of these implementations. We write the protocol plugins in C code and use the `clang` and `llvm` tools to produce the eBPF bytecode which can be injected inside our QUIC implementations. The eBPF virtual machine interacts with the QUIC implementation through a structure stored in the shared memory. To make implementations pluggable, we inserted (*i*) a map of plugins for each connection context with a dedicated API to inject code, (*ii*) an API implementing anchors, testing if a plugin is present to run it and processing the shared context, and (*iii*) code integrating the eBPF virtual machine as a library. Then, at applicative-level, hosts can communicate the plugins alongside with their identifiers over a dedicated QUIC data stream. The receiver can then inject the plugins by using the dedicated API. We perform our experiments in the Mininet environment [29].

3.1 Protocol Plugins for `picoquic`

As both `picoquic` and eBPF plugins are written in C, we use a C structure to implement the shared data structure. More precisely, when a plugin executes, it receives a pointer to a dedicated memory area that it can directly access and modify. We consider two use cases with `picoquic`. The first is Tail Loss Probe (TLP) [16]. TLP is a TCP extension that improves the performance of interactive applications when

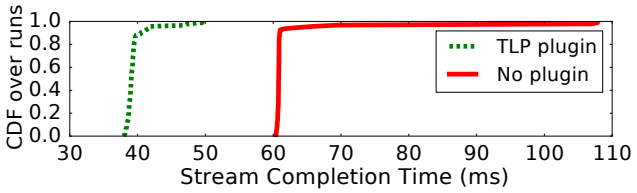

Figure 3: Adding plugins to implement TLP helps to reduce latency if the packet opening the stream is lost.

the last packet of a burst is lost. Our second use case is adding to QUIC support for Explicit Congestion Notification (ECN) [18].

3.1.1 A Protocol Plugin for Tail Loss Probe . Flach et al. [16] showed that small loss recovery additions can improve protocol performance. In particular, the proposed *Reactive* approach aims to alleviate the retransmission timeouts at client side when request tail losses occur. This algorithm, also known as Tail Loss Probe (TLP), can be enabled in Linux devices through `sysctl`.

To ensure this behaviour over the connection, we develop the TLP extension injected to the client by the server. Our extension improves the retransmission strategy of `picoquic` by retransmitting the last outstanding packet when the timer set up by the *Reactive* approach [16] fires. As suggested by Fig. 1, this extension instruments two pluggable operations: *SetTimer* is modified to also compute the TLP timer and *SelectRetransmission* now checks for expirations of the TLP timer to quickly retransmit the last outstanding packet. This behaviour is implemented in 81 lines of C code[1] which are compiled into 2.6 KB of eBPF bytecode.

Our server pushes the TLP extension to the client after the QUIC handshake. To demonstrate the benefits of this plugin, we consider a simple Mininet [29] scenario. The client uses the QUIC connection to send short requests that trigger short answers by the server. Each request is sent in a dedicated QUIC stream and the network exhibits a 10 ms round-trip-time. We then compute the completion time of a stream carrying one request and one response packet, when the first request packet is always lost. Figure 3 shows the CDF of these completion times over 100 runs. As expected, the request/response exchanges complete faster with the TLP strategy. With the *Reactive* approach and a single outstanding packet, the timer fires at $1.5 \times \text{RTT} + \text{WDT}$ while RTO fires at $\text{RTT} + 4 \times \text{RTT}_{\text{var}} + \text{WDT}$, with WDT being the worst-case delayed-ack timer. With the impact of the delayed-ack timer, it takes 30 ms to react with our extension, compared to 50 ms without. Our plugin therefore allows faster retransmissions of lost client requests.

3.1.2 Supporting ECN with a Protocol Plugin. The IETF QUIC working is currently debating on the best approach to include ECN inside QUIC [53]. We implemented in

[1] As these plugins replace built-in code, they also contain regular RTO timer computations. Only 36 lines actually implement TLP.

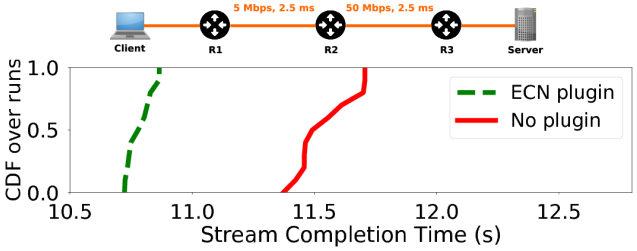

Figure 4: Thanks to the ECN plugin injected on the client by the server, the transfer time is reduced.

`picoquic` the July 2018 proposal which defines the `ACK_ECN` frame [43] to carry ECN feedback. This `ACK_ECN` frame is sent when a host receives a QUIC packet with the *Congestion Experienced* (CE) bit set. Our plugin is implemented in 152 lines of C code and is compiled into 5.6 KB. It modifies five pluggable operations in `picoquic`. *AfterPacketRead* to check whether the received packet had the CE bit set. *GenerateFrames* is extended to create the new `ACK_ECN` frame reporting the number of packets with the CE bit set. *InitOpaqueData* is an anchor run when the code is injected to initialise the opaque field to store the number of packets received with the CE bit set. *ParseFrames* is extended to parse the `ACK_ECN` frame. *AfterParseFrames* is modified to reduce the congestion window based on received `ACK_ECN` frame. These five operations implement the basic support for ECN in QUIC [43]. This plugin can be pushed by the server to the client at the beginning of the QUIC connection. Notice that successfully pushing the plugin allows the server to skip the ECN negotiation phase, as the injected code ensures client support.

To demonstrate the ECN plugin, we consider the network shown in the upper part of Fig. 4. Each router uses a buffer limited to 5 times the bandwidth-delay product (BDP) with tail drop. If ECN is activated, routers use the Random Early Detection queuing strategy [19] with ECN marking when the buffer exceeds $2 \times$ BDP. The server sends 5 MB over a single stream, and we repeat each experiment 10 times for each configuration. Figure 4 provides the CDF of the times to reliably transfer 5 MB. Without our ECN plugin, we configure the routers to drop packets when congestion occurs. In this case, `picoquic` takes 11.5 seconds in the median case to transfer 5 MB. With our ECN plugin, the server reacts quicker to congestion, decreasing the round-trip-time and retransmitted frames induced by packet buffering. This leads to roughly one second of spared completion time.

3.2 Protocol Plugins for `mp-quic`

We use `mp-quic` to demonstrate that *protocol plugins* can be pushed in an implementation written in a different language than C and explore multipath use cases. To enable the eBPF virtual machine to interact with `mp-quic`, we use `cgo` [24]. This standard `go` package allows to call C code (and by extension eBPF code) from a `go` program. Unlike

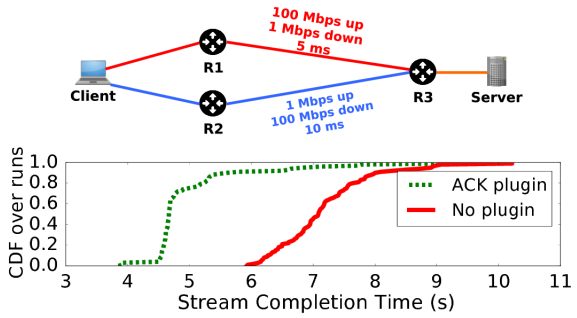

Figure 5: When plugin code force ACK frame sending on the R1-R3 path, the transfer completes quicker.

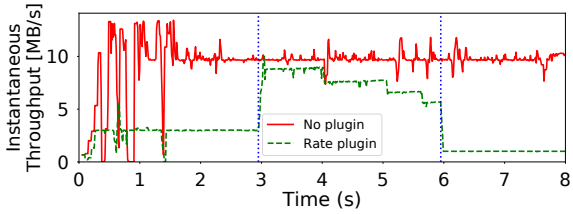

Figure 6: Three plugins controlling the rate over an expensive path.

`picoquic`, `mp-quic` requires data to be copied into a particular structure passed to `cgo` and eventually to the virtual machine.

3.2.1 Tuning ACKs with a Protocol Plugin. Our first multipath use case is a scenario with asymmetric bandwidth shown in the upper part of Fig. 5. In this environment, it would be beneficial to send the data over the bottom path and the acknowledgements over the upper one. This is possible with QUIC since frames are independent of the packet carrying them [12] in contrast with Multipath TCP [20, 44] where acknowledgements must be sent on the same path as data. However, as stated in the QUIC specification [9], changing the acknowledgement strategy can affect performance since latency measurements make some assumptions on when acknowledgements are generated. *Protocol plugins* prevent this ambiguity since a host can push the plugin that implements its chosen strategy.

We consider the network shown in Fig. 5, where both paths are asymmetric, with a 20 MB bulk download. Sending acknowledgements on the same path as data is not the best strategy, as acknowledgements would saturate the R2-R3 link. With this strategy, our Mininet experiments (plain curve of Figure 5) show that the client needs between 6 and 10 seconds to download the 20 MB file from the server. We then implement a plugin that attaches to the *SelectACKPath* operation on the client to force it to send all acknowledgements on the upper path. The dotted curve of Figure 5 shows that this strategy significantly reduces the download times as acknowledgements are returned quickly to the server and they do not saturate the upper link.

3.2.2 Restricting the Pacing Rate. Our last use case is inspired by smartphones. Multipath protocols such as Multipath TCP [20] enable them to simultaneously use both the WiFi and the cellular network. This enables fast handovers which is the main reason why Apple uses Multipath TCP on their smartphones [4]. However, many users have volume caps on their cellular plans and don't want to use the cellular network for large downloads when WiFi is available.

In this situation, the smartphone needs to control the operation of the server to restrict the bandwidth consumed on the cellular network. However, a server using a multipath protocol cannot easily determine whether a given path reaches a smartphone through its cellular or its WiFi interface. We address this problem with a *protocol plugin* that is pushed by the smartphone to the server at the beginning of the connection. This plugin is attached to the congestion control mechanism on the server and throttles its congestion window over the cellular path but not over the WiFi one. By writing 10 lines in C, our 0.5 KB plugin allows this behaviour. Figure 6 shows the effect of three plugin versions over a two-path scenario with 80 Mbps 50 ms RTT links. The first one, injected after the handshake, limits the rate to 3 MB/s. The second one, inserted after three seconds, caps the speed to 10 MB/s. The third one, added after six seconds, constraints the link to 1 MB/s. Our results presented in Fig. 6 confirm the effect of these plugins and demonstrate that plugins can be injected at any time on a connection after the handshake.

3.3 The Cost of Protocol Plugins

The proposed *protocol plugins* come with a trade-off between customisation and performance. Two factors influence their performance. First, there is a network overhead to exchange protocol plugins on a per connection basis. Second, there is some computational overhead to call plugin code instead of native one.

Network Overhead. The first overhead of *protocol plugins* is the transmission of the executable code. Table 2 shows the size of the bytecode of the plugins for the use cases studied in this section. With the eBPF user-space virtual machine used for our experiments, the executable codes are ELF files. The plugin sizes rarely exceed a few KB. This overhead is negligible for video streaming applications, or the download of large webpages over HTTP/2 or QUIC. Our largest plugin, the ECN one that is attached to five anchors only consumes 5.6 KB which is comparable to TLS certificate chains [45]. The `picoquic` use cases require larger ELF files because they are more complex than the `mp-quic` ones and need to be attached to several operations.

Performance Overhead. Executing the *protocol plugin* inside a virtual machine provides portability and isolation, but is slower than native code. Furthermore, the interactions with the instrumented implementation can also introduce some overhead. To quantify this overhead, we benchmark our implementations by measuring the throughput on localhost when

BPF Code	CGO noop	BPF noop	Preparing structure	Empty structure to CGO	Empty structure to BPF
-25.1 %	-4.9%	-6.1 %	-8.9 %	-21.7 %	-21.7 %

Table 1: `mp-quic` performance difference with relation to native Go code.

Use case	TLP	ECN	ACK Path	Pacing Rate
Size (KB)	2.6	5.6	1.0	0.5

Table 2: Network overhead of plugins for each studied use case.

transferring 50 MB. This is a standard benchmark used by the `quic-go` implementation.

We first consider the `picoquic` implementation and the TLP plugin, executed after each packet sent. Over 90 runs, we observed that both with and without the TLP plugin, `picoquic` reaches roughly 160 Mbps,[2] showing no significant impact from the plugin.

We now consider the `mp-quic` implementation. The packet scheduler is a critical component for multipath protocols [39] and previous works proposed to make it flexible to application needs [21]. We consider the lowest-latency packet scheduler, both in native Go code (no plugin) and as a plugin. This plugin weights around 3 KB for 107 lines of C code. The scheduler is called for each packet to be sent. Table 1 indicates that in `mp-quic`, this plugin reduces the achieved throughput by 25% compared to its native Go variant. As described in Sect. 3.2, `cgo` bridges the Go implementation with the eBPF VM. To communicate with the eBPF VM, `mp-quic` must first prepare a structure that exposes the connection context to the VM. As `mp-quic` cannot predict which fields of the connection context will actually be used it must put all of them in the structure. This consumes CPU time. If the scheduler in native Go code also fills such structure, performance drops by 9%. However, the main overhead comes from the stack switch induced by `cgo`. Its impact depends on the size of the interface structure. Running native Go scheduler with a `cgo` no-op without any communicated data decrease the performance by 5%. Doing the same with a 8 KB communicated structure lowers the performance by 22%. In such cases, we do not observe any difference between `cgo` and BPF no-ops.

4 DISCUSSION

In this paper, we have proposed and implemented *protocol plugins*, allowing transport protocol to be customisable on a per-connection basis. We also provided early results with several use cases showing the feasibility of such design with two implementations written in C and Go. However, our approach raises open questions.

Which protocols can benefit from plugins? Our results show that QUIC can obviously be extended and tuned by using *protocol plugins*. Beyond QUIC, would a similar approach work in the network, transport, application layer and for control-plane protocols? The past experience with active networks [7] shows that this does not fit with connectionless network protocols. Remotely injected TCP plugins also appear difficult given the prevalence of middleboxes. For the other protocols, integrity protection, e.g., with TLS, seems to be an important prerequisite for *protocol plugins*.

How should we specify a protocol that supports plugins? Assume that the next version of QUIC includes *protocol plugins*. Should it be specified as a small set core operations and leave the other features as plugins, or as a large set of operations and only use plugins to finely tune these operations? How do we specify the virtual machine, the *pluggable operations* and the API that any implementations needs to expose? How can we test the interoperability of such implementations?

How can we verify the safety of plugins? Remotely injected code raises obvious security concerns. Hosts should be able to verify the validity of the plugins that they received before running them. A first approach would be to use code-signing and trusted entities that certify plugins. Another approach would be to use verification techniques to validate the received bytecode. In all cases, each *pluggable operation* should have verifiable safety and liveness properties to prevent, e.g., a client from disabling the server congestion control algorithm by injecting malicious plugins.

What are the system implications ? Our prototype uses the eBPF virtual machine. This execution environment works, but other environments might provide better performance, stronger isolation properties or could interact more easily with implementations written in other langages than C. Can we leverage the eBPF support on smart NICs ? Can we apply a similar approach to in-kernel implementations and user-space implementations that leverage DPDK, XDP or similar APIs?

REFERENCES

[1] N. Amit and M. Wei. The design and implementation of hyperupcalls. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 2018.

[2] Anonymous. ebpf protocol plugins. Available via the Chairs., 2018.

[3] A. Begel, S. McCanne, and S. L. Graham. Bpf+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 123–134. ACM, 1999.

[4] O. Bonaventure and S. Seo. Multipath tcp deployments. *IETF Journal*, 12(2):24–27, 2016.

[5] L. Brakmo. TCP-BPF: Programmatically tuning TCP behavior through BPF. NetDev 2.2, 2017.

[6] L. S. Brakmo and L. L. Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *IEEE Journal on selected Areas in communications*, 13(8):1465–1480, 1995.

[7] K. Calvert. Reflections on network architecture: an active networking perspective. *ACM SIGCOMM Computer Communication Review*,

[2] This is roughly four times slower than vanilla `mp-quic`.

36(2):27–30, 2006.

[8] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. TCP Fast Open. RFC 7413, Dec. 2014.

[9] Q. D. Coninck and O. Bonaventure. Multipath Extension for QUIC. Internet-Draft draft-deconinck-quic-multipath-00, Internet Engineering Task Force, Mar. 2018. Work in Progress.

[10] J. Corbet. Uprobes in 3.5. https://lwn.net/Articles/499190/, 2012.

[11] Q. De Coninck. mp-quic. https://github.com/qdeconinck/mp-quic, 2017.

[12] Q. De Coninck and O. Bonaventure. Multipath quic: Design and evaluation. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 160–166. ACM, 2017.

[13] M. Duke, R. Braden, W. Eddy, E. Blanton, and A. Zimmermann. A Roadmap for Transmission Control Protocol (TCP) Specification Documents. RFC 7414, Feb. 2015.

[14] J. Edge. A seccomp overview. *Linux Weekly News*, September 2015. https://old.lwn.net/Articles/656307/.

[15] K. R. Fall and W. R. Stevens. *TCP/IP illustrated, volume 1: The protocols*. addison-Wesley, 2011.

[16] T. Flach et al. Reducing web latency: the virtue of gentle aggression. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 159–170. ACM, 2013.

[17] M. Fleming. A thorough introduction to ebpf. *Linux Weekly News*, December 2017. https://old.lwn.net/Articles/740157/.

[18] S. Floyd. Tcp and explicit congestion notification. *ACM SIGCOMM Computer Communication Review*, 24(5):8–23, 1994.

[19] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on networking*, 1(4):397–413, 1993.

[20] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824, Jan. 2013.

[21] A. Frömmgen et al. A programming model for application-defined multipath tcp scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 134–146. ACM, 2017.

[22] K. Fukuda. An analysis of longitudinal tcp passive measurements (short paper). In *International Workshop on Traffic Monitoring and Analysis*, pages 29–36. Springer, 2011.

[23] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot. Vmkit: a substrate for managed runtime environments. In *ACM Sigplan Notices*, volume 45, pages 51–62. ACM, 2010.

[24] Go. Command `cgo`. https://golang.org/cmd/cgo/.

[25] B. Gregg. ebpf: One small step. http://www.brendangregg.com/blog/2015-05-15/ebpf-one-small-step.html, May 2015.

[26] B. Gregg and J. Mauro. *DTrace: dynamic tracing in oracle Solaris, Mac OS X and freeBSD*. Prentice Hall Professional, 2011.

[27] S. Ha, I. Rhee, and L. Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[28] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. In *ACM SIGPLAN Notices*, volume 52, pages 185–200. ACM, 2017.

[29] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *CONEXT'12*, pages 253–264. ACM, 2012.

[30] M. Honda et al. Is it still possible to extend tcp? In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 181–194. ACM, 2011.

[31] C. Huitema. picoquic. https://github.com/private-octopus/picoquic, 2018.

[32] IO Visor Project. Userspace ebpf vm. https://github.com/iovisor/ubpf, 2018.

[33] J. Iyengar and M. Thomson. Quic: A udp-based multiplexed and secure transport. *draft-ietf-quic-transport-01 (work in progress)*, 2017.

[34] A. Langley et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 183–196. ACM, 2017.

[35] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[36] J. Nagle. Congestion control in ip/tcp internetworks. *ACM SIGCOMM Computer Communication Review*, 14(4):11–17, 1984.

[37] A. Narayan, F. Cangialosi, P. Goyal, S. Narayana, M. Alizadeh, and H. Balakrishnan. The case for moving congestion control out of the datapath. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 101–107. ACM, 2017.

[38] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. *Practical dynamic software updating for C*, volume 41. ACM, 2006.

[39] C. Paasch, S. Ferlin, O. Alay, and O. Bonaventure. Experimental evaluation of multipath tcp schedulers. In *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop*, pages 27–32. ACM, 2014.

[40] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988, Nov. 2000.

[41] V. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP's Retransmission Timer. RFC 6298 (Proposed Standard), June 2011.

[42] J. Postel. Transmission Control Protocol. RFC 793, Sept. 1981.

[43] QUIC Working Group. Ecn in quic. https://github.com/quicwg/base-drafts/wiki/ECN-in-QUIC, 2018.

[44] C. Raiciu et al. How hard can it be? designing and implementing a deployable multipath tcp. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 29–29. USENIX Association, 2012.

[45] J. Rüth, C. Bormann, and O. Hohlfeld. Large-scale scanning of tcp's initial window. In *Proceedings of the 2017 Internet Measurement Conference*, pages 304–310. ACM, 2017.

[46] J. Rüth, I. Poese, C. Dietzel, and O. Hohlfeld. A first look at quic in the wild. In *International Conference on Passive and Active Network Measurement*, pages 255–268. Springer, 2018.

[47] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: network processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012.

[48] R. Stewart (Ed.). Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), Sept. 2007.

[49] B. Trammell et al. Tracking transport-layer evolution with pathspider. In *Proceedings of the Applied Networking Research Workshop*, pages 20–26. ACM, 2017.

[50] V. Vasudevan et al. Safe and effective fine-grained tcp retransmissions for datacenter communication. In *ACM SIGCOMM computer communication review*, volume 39, pages 303–314. ACM, 2009.

[51] K. Wang et al. Draining the swamp: Micro virtual machines as solid foundation for language development. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[52] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *OSDI*, pages 33–47, 2014.

[53] M. Westerlund. Proposal for adding ecn support to quic. https://github.com/quicwg/base-drafts/pull/1372, 2018.