# Poster: Multipath Extensions for WireGuard

Konrad-Felix Krentz[1] and Marius-Iulian Corici
Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany
konrad.krentz(at)it.uu.se, marius-iulian.corici(at)fokus.fraunhofer.de

***Abstract*—The tunneling protocol WireGuard outperforms its main competitors OpenVPN and IPsec in terms of throughput and latencies. These improvements are due to WireGuard's use of faster crypto primitives, as well as to the implementation of WireGuard as a Linux kernel module that uses multithreading and advanced locking strategies. Independently of the WireGuard project, Lukaszewski et al. demonstrated improvements in end-to-end goodput when tunneling protocols exploit alternative communication paths. In this poster, we combine these two research directions by proposing multipath extensions for WireGuard. Our extensions involve additions to the WireGuard header, which enable obtaining real-time statistics on the performance of each path. Further, these real-time path performance statistics enable a self-adaptive selection of paths. As a proof of concept, we adapted the WireGuard Linux kernel module accordingly and prototyped four example path schedulers, two of which adopt multi-armed bandit algorithms.**

***Index Terms*—WireGuard, MPTCP, multipath, self-adaptive**

## I. Introduction

Tunneling protocols serve for relaying packets between physically distant networks. Well-known instances of such protocols are OpenVPN and IPsec. Both come with security features that ensure the confidentiality, integrity, and authenticity of the relayed packets. WireGuard is a modern alternative to OpenVPN and IPsec [1]. Compared to OpenVPN and IPsec, WireGuard provides higher performance, better user-friendliness, as well as stronger security [1].

Independently of the WireGuard project, Lukaszewski et al. conducted a series of experiments to test whether end-to-end goodput improves when tunneling protocols exploit alternative communication paths [2]. In their experiments, a user downloaded files from an Apache2 web server over a tunnel that can use two paths. The tunnel's underlying transport protocol varied across experimental runs. Lukaszewski et al. tested the Transmission Control Protocol (TCP), Multipath TCP (MPTCP), the User Datagram Protocol (UDP), and Multipath UDP (MPUDP) - a version of UDP that balances datagrams across all available paths according to a configured split ratio. The main results of their experiments are (i) that MPTCP consistently achieves better end-to-end goodput than TCP, (ii) that MPUDP consistently achieves better end-to-end goodput than UDP, and (iii), most interestingly, that MPUDP achieves better end-to-end goodput than MPTCP if the configured split ratio is manually adjusted to the current round trip times (RTTs) of both paths, just like the path scheduler of the tested MPTCP implementation does.

[1] Dr. Krentz is now with Uppsala Universitet

Annex to ISBN 978-3-903176-39-3© 2021 IFIP

While WireGuard lacks native multipath support, one workaround is to tailor Lukaszewski et al.'s MPUDP to WireGuard. That is, WireGuard's outgoing packets, all of which are UDP datagrams, can be balanced across all available paths, e.g., according to a static split ratio. However, Lukaszewski et al. showed, balancing packets as per a static split ratio yields suboptimal results in terms of end-to-end goodput. Rather, the split ratio should continuously be adapted to the current performance of the available paths. Yet, real-time path performance statistics are not readily available to WireGuard peers. This is because WireGuard is based on UDP, which does not send acknowledgments. Conversely, in MPTCP, each segment is being acknowledged, which enables computing various real-time path performance statistics, such as RTTs. This is actually what allows MPTCP path schedulers to self-adapt [3].

Another workaround to add multipath support to WireGuard builds on MPTCP. The idea is to set up one WireGuard tunnel per available path. These WireGuard tunnels can then be treated as paths by an MPTCP-based tunneling protocol. The main advantage over the above MPUDP-based workaround is that existing MPTCP path schedulers can be used to balance segments across all available paths in a self-adaptive manner. But, this second workaround has three drawbacks. First, it incurs a high communication overhead. MPTCP, e.g., acknowledges each segment and each WireGuard tunnel additionally creates its own control traffic. Second, when tunneling TCP over MPTCP, TCP-over-TCP issues arise [2]. Finally, switching from UDP to TCP leads to latency degradations, too [4].

The main contribution of this poster is to propose multipath extensions for WireGuard. Unlike the workarounds described above, our multipath extensions intimately integrate into WireGuard. This avoids the overhead and performance penalties of the MPTCP-based workaround, while also avoiding the lacking self-adaption of the MPUDP-based workaround. In the following, we describe the design of our multipath extensions for WireGuard and sum up first experimental results. For space reasons, we assume familiarity with the WireGuard protocol.

## II. Design

Our multipath extensions for WireGuard are organized into a data and a control plane, as shown in Fig. 1. The data plane mainly collects real-time statistics on the performance of each path. Remarkably, these statistics are gathered without sending acknowledgments or other extra messages. Besides, the data plane provides an interface for retrieving its statistics, as well as for selecting the path on which an outbound WireGuard

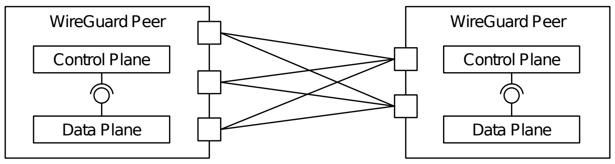

Fig. 1. Overview of our multipath extensions for WireGuard

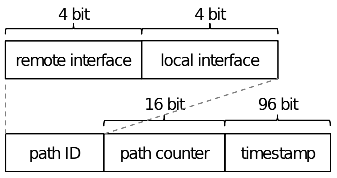

Fig. 2. Fields added to WireGuard messages[2]

message[2] is to be sent. The control plane interfaces with the data plane to tune the path selection.

## A. Data Plane

To obtain real-time path performance statistics, the data plane makes additions to the headers of WireGuard messages. One set of additions relates to enabling receivers to get real-time statistics on how well paths perform in the inbound direction. Another set of additions to the headers of WireGuard messages relates to enabling senders to get real-time statistics on how well paths perform in the outbound direction.

Concretely, the first set of additions consists of the following three new header fields in each WireGuard message[2], as shown in Fig. 2. The first new header field is named `Path ID`. It contains an 8-bit ID of the path on which the message is sent. The lower 4 bits identify the local network interface, while the upper 4 bits identify the remote network interface. Many paths may exist between two WireGuard peers, each of which has an individual `Path ID`, as shown in Fig. 1. The second new header field is named `Path Counter`. It contains a 16-bit rolling-over counter counting the overall number of WireGuard messages[2] that were sent on the path on which the message is sent. The third new header field is called `Transmission Time`. It contains a TAI64N timestamp of the time span between boot time and the moment of transmitting the message.

By means of these new header fields, the receiver side can infer a path's current throughput, delivery ratio, and "relative path latency". Throughput becomes measurable thanks to `Path ID`s. They inform the receiver side via which path a received WireGuard message[2] was sent. This hint can serve to compute the amount of data received via a certain path. Delivery ratios become measurable thanks to `Path Counters`. These incrementing counters allow the receiver side to notice if a WireGuard message[2] got lost. Further, by keeping track of missed and received `Path Counter`s, and by accommodating out-of-delivery, per-path delivery ratios can be calculated. Finally, the latencies of the available paths become comparable thanks to `Transmission Time`s. Using these timestamps, the receiver of a WireGuard message[2] can compute the "relative path latency" as the difference between the receiver's time since boot at the moment of receiving the message and the message's `Transmission Time`. Though the result deviates from the real path latency, it suffices for deducing how much faster or slower a path is than another.

[2] here, we only mean WireGuard messages of type `Handshake Initiation`, `Handshake Response`, and `Transport Data`

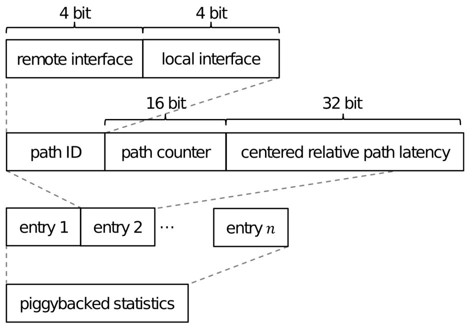

Fig. 3. Wire format of piggybacked real-time path performance statistics

The header additions described so far enable the receiver side to get real-time statistics on how well paths perform in the inbound direction. Next, we detail the second set of header additions, which report back on these statistics to the sender side, allowing senders to make informed decisions on which path to select even when paths perform asymmetrically.

For reporting back real-time path performance statistics, the data plane uses the fact that WireGuard sends replies to many messages anyway. Specifically, a `Handshake Initiation` message triggers a `Handshake Response`. Likewise, a `Handshake Response` triggers a `Transport Data` message, and a `Transport Data` message triggers, unless it is itself a reply, another `Transport Data` message. Consequently, the data plane piggybacks statistics it wishes to send back on `Handshake Response` and `Transport Data` messages.

The statistics that the data plane wishes to send back are buffered in a list. A new entry is appended to this list upon receiving a fresh authentic WireGuard message[2]. Such a new entry encompasses (i) the `Path ID` of the path on which the received message was sent, (ii) the path counter of the received message, (iii) the relative path latency subtracted by the first measured relative path latency so as to roughly center relative path latencies around zero, and (iv) an initially non-zero redundancy counter. The redundancy counter is decremented each time when the entry is sent back to the sender side. Once the redundancy counter of an entry reaches zero, the entry is deleted from the list. The purpose of retransmitting entries is to avoid, to a configurable extent, that the sender side falsely concludes that a message got lost, though it is actually the reply that got lost. The wire format of piggybacked statistics is shown in Fig. 3.

For security reasons, the data plane encrypts and authenticates all its header additions. Encrypting `Transmission Times` is critically important since sending them unencrypted would entail the risk of leaking side-channel information to attackers. Authenticating all header additions is also crucial because if attackers tampered with these header fields, this could result in suboptimal path selections.

As for `Transport Data` messages, header additions are encrypted and authenticated in the same way as the encapsulated packets. However, as for `Handshake Initiation` and `Handshake Response` messages, symmetric keys are not yet established when these messages are being sent. To this end, the data plane uses a temporary symmetric key that is available at this stage. This key is denoted by $\kappa$ in [1].

We note that $\kappa$ is not available when sending WireGuard messages of type `Cookie Reply`, which raises two subtleties. First, to encrypt and authenticate header additions to `Cookie Reply` messages, the data plane would have to generate such a key. This would, however, contradict the purpose of `Cookie Reply` messages, which is to avoid time-consuming computations until the initiator has made additional efforts. As a solution, the data plane forgoes using `Cookie Reply` messages for collecting real-time path performance statistics, as well as forgoes using them for reporting back on such statistics. Second, $\kappa$ is also not generated upon reception of a `Handshake Initiation` message if a `Cookie Reply` message is to be sent. To this end, the data plane ignores the header additions contained in `Handshake Initiation` messages in such cases. Subsequently, when the initiator retransmits the `Handshake Initiation` message containing the received cookie, that message must use the same path counter as the original `Handshake Initiation` message. This avoids that the initiator falsely concludes that the initial `Handshake Initiation` message got lost.

### B. Control Plane

Our prototyped control plane offers a choice between four different path schedulers and is extensible to support further path schedulers, as well. A path scheduler encapsulates the logic of selecting the paths via which outgoing `Handshake Initiation`, `Handshake Response`, and `Transport Data` messages are to be sent. `Cookie Reply` messages, on the other hand, are always sent via the same path as was used by the corresponding `Handshake Initiation` message. This obviates allocating memory for communication with unauthenticated peers, which would be security-critical.

Two basic path schedulers we prototyped are the round-robin (RR) and the minLatency path schedulers. The RR path scheduler, on the one hand, distributes outgoing WireGuard messages across all available paths in a round-robin fashion. The minLatency path scheduler, on the other hand, selects the path with the lowest sample mean of the relative path latencies in the outbound direction after an initial sampling period.

Both other path schedulers we prototyped treat path selection as a piecewise-stationary multi-armed bandit (MAB) problem [5], [6]. That is, paths correspond to arms, the path scheduler takes the role of the agent, and, after selecting a path for an outgoing WireGuard message, the path scheduler gets a reward in accordance with the path's performance. Specifically, we defined the reward for sending the $t$-th message via path $k$ as:

$$X_{k,t} = \begin{cases} 0, & \text{if } Y_{k,t} = 0 \\ \alpha + \beta\left(1 - \frac{1}{1+e^{-g(Z_{k,t} - z_{1,1})}}\right), & \text{else} \end{cases} \quad (1)$$

where:

- $Y_{k,t}$ is a Bernoulli random variable with value 1 if the message arrives and 0 otherwise
- $\alpha, \beta \in [0, 1], (\alpha + \beta = 1)$ are parameters for balancing priorities between delivery ratios and path latencies
- $Z_{k,t}$ is a random variable that represents the message's relative path latency
- $z_{1,1}$ is the first measured relative path latency and serves for centering relative path latencies roughly around zero
- $g$ is the logistic growth rate of the logistic function $\frac{1}{1+e^{-g(Z_{k,t} - z_0)}}$, which normalizes relative path latencies, as required by the MAB algorithms we chose [5], [6]

## III. Conclusions and Future Work

The tunneling protocol WireGuard is gaining popularity due to its performance, user-friendliness, and security. However, WireGuard lacks native multipath support and the described workarounds have limitations. To fill this gap, we have integrated multipath extensions into WireGuard. We have followed the software-defined networking paradigm so that path schedulers are exchangeable. This has paved the way for comparing various path schedulers in our preliminary evaluation. On the poster, we will present that preliminary evaluation. The results suggest that our sliding-window upper confidence bound (SW-UCB)-based path scheduler performs best in terms of latencies and delivery ratios [6]. For future work, we suggest (i) mitigating out-of-order delivery, which currently occurs when our MAB algorithm-based path schedulers explore unrewarding paths, (ii) investigating the use of only some of WireGuard's messages for collecting real-time path performance statistics, and (iii) considering contextual bandit algorithms for path scheduling so as to take contextual information into account.

## References

[1] J. A. Donenfeld, "Wireguard: Next generation kernel network tunnel," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS 2017)*. Internet Society, 2017.

[2] D. Lukaszewski and G. Xie, "Multipath transport for virtual private networks," in *Proceedings of the 10th USENIX Workshop on Cyber Security Experimentation and Test (CSET 17)*. USENIX, 2017.

[3] M. Polese, F. Chiariotti, E. Bonetto, F. Rigotto, A. Zanella, and M. Zorzi, "A survey on recent advances in transport layer protocols," *IEEE Communications Surveys Tutorials*, vol. 21, no. 4, pp. 3584–3608, 2019.

[4] I. Coonjah, P. C. Catherine, and K. M. S. Soyjaudah, "Experimental performance comparison between TCP vs UDP tunnel using OpenVPN," in *Proceedings of the 2015 International Conference on Computing, Communication and Security (ICCCS)*. IEEE, 2015, pp. 1–5.

[5] Y. Cao, Z. Wen, B. Kveton, and Y. Xie, "Nearly optimal adaptive procedure with change detection for piecewise-stationary bandit," in *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS)*. PMLR, 2019, pp. 418–427.

[6] A. Garivier and E. Moulines, "On upper-confidence bound policies for switching bandit problems," in *Proceedings of the International Conference on Algorithmic Learning Theory (ALT 2011)*. Springer, 2011, pp. 174–188.