TCPLS: Modern Transport Services with TCP and TLS

Florentin Rochet
University of Edinburgh
Scotland
frochet@ed.ac.uk

Emery Assogba
UCLouvain
Belgium
emery.assogba@uclouvain.be

Maxime Piraux
UCLouvain
Belgium
maxime.piraux@uclouvain.be

Korian Edeline
Université de Liège
Belgium
korian.edeline@gmail.com

Benoit Donnet
Université de Liège
Belgium
benoit.donnet@uliege.be

Olivier Bonaventure
UCLouvain
Belgium
olivier.bonaventure@uclouvain.be

ABSTRACT

TCP and TLS are among the essential protocols in today's Internet. TCP ensures reliable data delivery while TLS secures the data transfer. Although they are very often used together, they have been designed independently following the Internet layered model. This paper demonstrates the various benefits that a closer integration between TCP and TLS would bring.

By leveraging the extensible TLS 1.3 records, we combine TCP and TLS into TCPLS to build modern transport services such as multiplexing, connection migration, stream steering, and bandwidth aggregation. These services do not modify the TCP wire format and are resistant to middleboxes. TCPLS offers a powerful API enabling applications to precisely express the required transport services, ranging from a single-path single-stream connection to a multi-stream connection over several network paths, enabling choices between aggregated bandwidth and head-of-line blocking avoidance.

Compared to MPTCP, our TCPLS prototype offers more control to the application and can be easily deployed as an extension to user-space TLS libraries, while being implemented at a low cost. Measurements demonstrate that it offers higher performance than existing QUIC libraries with a super set of transport services.

CCS CONCEPTS

• **Networks → Transport protocols**; **Network protocol design**.

ACM Reference Format:
Florentin Rochet, Emery Assogba, Maxime Piraux, Korian Edeline, Benoit Donnet, and Olivier Bonaventure. 2021. TCPLS: Modern Transport Services with TCP and TLS. In *The 17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21), December 7–10, 2021, Virtual Event, Germany.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3485983.3494865

1 INTRODUCTION

The Transmission Control Protocol (TCP) [80] is one of the most critical protocols in today's Internet. It has been designed following a layer approach and now serves a wide range of applications. During the last four decades, TCP evolved under the pressure of competing protocols. In the 1980s, software-based TCP implementations were considered too slow. Researchers proposed new transport protocols such as XTP [90] which could be implemented in hardware. Meanwhile, TCP implementations got a considerable speed boost [19] and XTP disappeared. The TCP speed boost and usage triggered the development of various important TCP extensions, including timestamps and large windows [14] or Selective Acknowledgments [65].

In the mid-nineties, the Secure Socket Layer (SSL) protocol was proposed as an additional layer to TCP to secure emerging e-commerce websites [42]. SSL evolved in different versions of the Transport Layer Security (TLS) protocol, the most recent one being version 1.3 [84]. Nowadays, TLS is almost ubiquitous on web servers [45] and many non-web applications use it [5].

During the late nineties, early 2000s, transport protocol researchers explored alternatives to TCP. The IETF standardized two new transport protocols: DCCP [55] and SCTP [95]. We rarely use DCCP today. Despite SCTP benefits (support for multihoming, better design, and extensibility), only niche applications use it. This limited deployment is mainly due to the various middleboxes (NAT, firewalls, etc.) deployed on the Internet often blocking packets that do not carry TCP or UDP [46]. SCTP initially supported multihoming by switching from one path to another. It was later extended to use different paths continuously [49]. Multipath TCP [28, 82] brought similar capabilities to TCP, and included a coupled congestion control scheme [109], later brought to SCTP as well. This particular succession of events shows how different designs compete and advance each other.

Extending TCP today is not feasible anymore as middleboxes severely interfere with changes to the TCP header and options [24, 46, 67]. To overcome this problem, Google started QUIC as an experimental protocol [56, 89] combining functions usually found in TCP, TLS, and HTTP/2. During the last years, it evolved into a complete transport protocol [50]. QUIC leverages encryption to prevent middlebox interference and propose to revisit the layered model of the Internet to improve the transport services. As QUIC runs atop UDP, it can be implemented and deployed as a user-space library.

Does the standardization of QUIC mark the end of the TCP era, moving all applications and transport research to QUIC? We do not think so. Today, QUIC is mainly used for HTTP/3 [101] and TCP remains a fallback because of its greater support in networks. TCP also still serves many applications [27, 103]. In the light of those recent advances, we revisit how transport services can be provided with TCP and TLS today. The QUIC design integrates services that were found in the security and application layers, e.g., encryption and multiplexing. TCP and TLS have been both designed in strict layers separating the two. This paper revisits this separation through the lens of the following research questions:

RQ1 - How can TCP and TLS be combined to improve extensibility and middlebox resilience ?

RQ2 - What are the new transport services that this combination can offer?

To answer these questions, we design and implement an approach that combines TCP and TLS 1.3 into a fast, flexible, and secure transport protocol called **TCPLS** [1]. At the heart of our approach, we illustrate how TLS records (i.e., messages exchanged through TLS) can be leveraged to build new transport services. When TLS is used over TCP, TLS AppData records are solely used to securely convey the TCP bytestream. In TCPLS, we extend their use to convey both application and control data including encrypted TCP options.

We demonstrate that this combination allows secure extensibility that can also be used together with techniques such as TCP Fast Open [18] to lower the handshake latency. We leverage this new extensibility to implement modern transport services such as multiplexing, connection migration, and stream steering capabilities without risking middlebox interference. Our TCPLS prototype is implemented as a user-space library exposing a powerful API to applications while leveraging the high-performance Linux kernel TCP stack. Our lab measurements indicate that TCPLS can be implemented at a low cost while providing higher bulk throughput and features than the QUIC implementations we tested.

The rest of this paper is organized as follows: Sec. 2 provides the required technical background; Sec. 3 discusses how we designed TCPLS, while Sec. 4 focuses on how we implemented TCPLS. Sec. 5 evaluates the performance and behavior of TCPLS. Sec. 6 discusses the related work. Finally, Sec. 7 concludes this paper by summarizing its main achievements and discussing further directions.

2 BACKGROUND

TCP was designed as an end-to-end protocol that is only used on end-hosts. TCP includes flow-control, supports various retransmission techniques to cope with packet losses, and congestion control mechanisms. TCP stacks have evolved a lot during the last decades. Most extensions to TCP leverage the TCP Options space, which is limited to 40 bytes. Unfortunately, the TCP designers did not foresee that many TCP extensions would be standardized. Today, the TCP header size is a constraint. The IETF has discussed this problem for several years, but the latest attempt to solve it [98] has not yet been implemented by major TCP stacks.

Modern applications rarely use TCP alone. They often combine TCP with Transport Layer Security (TLS). TLS 1.3 [84] brings several essential features compared to the previous versions. It includes a secure handshake that allows negotiating the security parameters and keys within one round-trip time. Thanks to TCP Fast Open [81], it is also possible to perform the secure handshake during the TCP handshake. Furthermore, it is also possible to exchange application data during the handshake. The TLS 1.3 record layer protects all application data with encryption and authentication. This record layer is extensible, and the TLS record types are also encrypted to prevent ossification.

[1] A preliminary version of this work has been presented in a workshop paper [87].

	TCP	MPTCP	TLS/TCP	QUIC	TCPLS
Reliability & cong. control	✓	✓	✓	✓	✓
Message conf. and auth.	✗	✗	✓	✓	✓
Failover	✗	✓	✗	(✓)	✓
HoL blocking avoidance	✗	✗	✗	✓	(✓)
Streams	✗	✗	✗	✓	✓
Connection migration	✗	(✓)	✗	(✓)	✓
Concurrent paths	✗	✓	✗	✗	✓

Table 1: Comparison of ✗ the services not offered, (✓) partially available, and ✓ offered by protocols.

Despite the Internet layered architectural principle, network operators have deployed a variety of middleboxes (e.g., firewalls, NATs, transparent proxies) [94] that sometimes interfere with TCP or its extensions [24, 46, 67]. Several types of middlebox interference have been identified by researchers and network engineers. The most relevant ones for TCP are: (*i*) Network Address Translators (NAT) which change IP addresses, port numbers and also the TCP checksum [106], (*ii*) Application Level Gateways which modify the TCP payload [43], (*iii*) firewalls that discard packets containing TCP Options that were not known when the firewall was designed [25], (*iv*) firewalls that replace unknown TCP Options with the NOP TCP Option, (*v*) transparent proxies which terminate TCP connections [107], and (*vi*) high-speed network adapters that fragment large TCP packets in a series of smaller packets and reassemble them [46]. All these in-path network functions make assumptions about the TCP packets content, invalidating the TCP end-to-end paradigm. Moreover, they do not always strictly follow the TCP specifications [41, 46], which may negatively affect TCP's evolution and performance [25]. Furthermore, deployed middleboxes remain inside the network for several years and sometimes up to a decade. Many of the older middleboxes are not regularly updated. These middleboxes severely limit the TCP extensibility. Multipath TCP [29, 82] (MPTCP) managed to cope with the interferences listed above, but at the price of increased complexity, most notably the utilization of a second level of checksum to detect middlebox interference [41, 82], and fallback mechanisms. Choosing another transport is not a deployable approach as many middleboxes block transport protocols other than TCP and UDP [8].

Google took a different approach at solving this problem by developing an entirely new secure transport protocol, QUIC [56], combining TCP, TLS, and HTTP features in a single protocol implemented in a userspace library and running above UDP. QUIC prevents middlebox interference by encrypting and authenticating the data, but also most of the control information such as the acknowledgments except a very small header. QUIC leverages TLS 1.3 [84] to negotiate the security keys and authenticate the server. This results in latency reduction, more security, and better extensibility. QUIC has a flexible framing system and packets are encrypted separately, similarly to TLS records. QUIC allows the application to

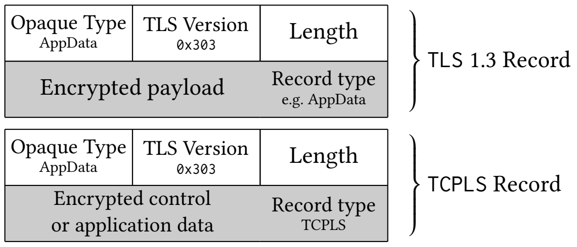

Figure 1: TCPLS extends the encrypted TLS records to convey control and application data.

use different datastreams over a single connection such as SCTP [95] or Structured Streams [30]. QUIC has been adopted by the industry and standardized within the IETF [50]. There are more than a dozen QUIC implementations [34, 63]. QUIC is already used in production as shown by recent measurement studies [102].

Table 1 summarizes the main features of TCP, MPTCP, TLS over TCP (TLS/TCP), and QUIC. All protocols include reliability and congestion control. TLS and QUIC provide the same security features. MPTCP efficiently supports failovers and QUIC includes a connection migration capability. QUIC supports streams and prevents Head-of-Line (HoL) blocking. MPTCP is the only standardized transport protocol that is able to aggregate the bandwidth of several paths. MPTCP implementations include several path management strategies [39, 40] to control the different paths utilization. QUIC allows the client to migrate its connection.

3 TCPLS DESIGN

The slow pace of innovation in TCP-based transport services can be explained by several factors. First, the amount of bytes for extensions in the TCP header is limited to 40 bytes. Second, TCP is often implemented as a part of the OS kernel, which adds complexity to implement and deploy any modification. Third, it is exchanged in cleartext and is known to be heavily ossified due to middleboxes, strongly restraining changes in the header semantics and to TCP Options. Following the recent advances in the transport layer, we go back to *RQ1* and ask *How can TCP and TLS be combined to improve extensibility and middlebox resilience* ?

We answer *RQ1* by leveraging the flexible encrypted TLS records to supplement the TLS-encrypted application data with transport-level control data allowing TCPLS to provide a large range of new transport services. In addition, existing TCP Options can be securely conveyed in new TLS records. These records are indistinguishable on the wire from TLS 1.3 AppData records, ensuring middlebox compatibility.

In this section, we explain how we combine TCP and TLS into TCPLS. In particular, Sec. 3.1 first introduces the TCPLS records which are the basic unit of communication in TCPLS. Sec. 3.2 explains how a TCPLS session is managed. Sec. 3.3 then shows what are the new transport services this combination can offer (*RQ2*) by showing how TCPLS uses TCPLS records to provide modern transport services such as multiplexing, connection migration, and a novel multipath approach.

3.1 TLS Records and Extensions

Each protocol has a basic unit for exchanging application and control data. For instance, TCP segments are split in two with only the header dedicated to control data. In QUIC, the basic unit is a QUIC packet which can carry several and different types of QUIC frames. TLS supports records of different types and lengths. As illustrated in Fig. 1, TCPLS leverages the TLS encrypted records that hide their true type and content from middleboxes in opaque AppData records. In this way, TCPLS can extend the messages exchanged with TLS in a secure and flexible manner.

We chose to design the TCPLS framing at the TLS record level so that decryption operations can be performed with a zero-copy codepath within a contiguous memory buffer, unlike QUIC. Indeed, any TCPLS control information not relevant to the application is located at the end of the record so that, by adjusting the offset when decrypting the TCPLS records, the receiver can overwrite them and achieve zero-copy within a contiguous allocated memory. This specific protocol design choice allows TCPLS implementations to easily offer zero-copy data delivery through a simple API. In comparison, QUIC's design forces developers to achieve zero-copy by offering a more complex API delivering potentially fragmented data.

TLS also offers a mean for requesting extended functionalities in a TLS session through request/response messages called TLS Encrypted Extensions. TCPLS leverages these messages to negotiate during the handshake some of the new transport services it offers.

TCPLS uses these records and extensions to exchange TCP Options. They can be negotiated at the start of the connection using TLS extensions or be exchanged later throughout the connection in encrypted TCPLS records. Since TLS messages are conveyed in the TCP payload, they are not space-constrained as existing TCP options. They are also exchanged reliably, which is a requirement for several TCP Options including TCP User Timeout option [33], MPTCP's ADD_ADDR and REMOVE_ADDR option and some experimental TCP options [97]. They also cannot be modified by middleboxes as they are secured by TLS. However, TCP proxies could prevent an end-to-end TCP connection and only exchanging the TCP bytestream, in which case this mechanism should not be used. Detecting TCP proxies is easy as one could echo a part of the TCP header in a TCPLS record to detect them.

3.2 Starting a TCPLS Session

A TCPLS session starts with a TCP handshake followed by a TLS handshake. This process can be safely accelerated through the use of TCP Fast Open [18]. A TCPLS client indicates its willingness to use TCPLS with the TCPLS Hello extension in its TLS ClientHello message. Upon reception of this extension, the TCPLS server replies with a TLS ServerHello message echoing the TCPLS Hello extension in TLS EncryptedExtensions. These are encrypted with the handshake key which is not part of the context used to derive the eventual application key. Once the TLS handshake has succeeded, the endpoints can exchange TCPLS records.

Our design ensures that middleboxes cannot distinguish TCPLS from TLS past the handshake. This eases the TCPLS deployment and hardens censorship attempts. Whenever a middlebox blocks the client TCPLS's handshake extensions, the later could choose

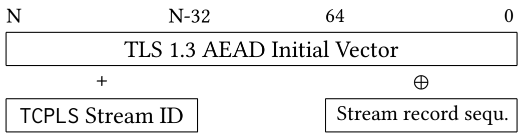

Figure 2: The AEAD Nonce of TCPLS Streams is derived from TLS 1.3.

to enable TCPLS after the handshake using TCPLS control records, provided that the server announced TCPLS support. The censor would not be able to distinguish this behaviour from regular TLS, hardening potential attempts for censoring applications using TCPLS. The server handshake extensions are encrypted and part of the TLS handshake transcript hash, and thus cannot be dropped by the censor without breaking the TLS handshake for all TLS clients.

3.3 TCPLS Transport Services

By leveraging TCPLS records and extensions, we can design new modern transport services atop the combination of TCP and TLS. In this subsection, we answer our second research question (*RQ2*) by presenting three transport services: stream multiplexing (Sec. 3.3.1), connection migration (Sec. 3.3.2), and bandwidth aggregation (Sec. 3.3.3). We leverage the TCPLS records to multiplex concurrent encrypted bytestreams. By extending the TLS handshake, TCPLS allows joining several TCP connections to a single TCPLS session to support connection migration, stream steering and bandwidth aggregation. These features are built atop the TCP bytestream, and as such they avoid middleboxe interferences that modify the TCP header.

3.3.1 Stream Multiplexing. Stream Multiplexing is a transport service that has been part of recent protocols such as SCTP [95], QUIC [50] and HTTP/2.0 [9]. The latter also proposes multiplexing with an independent framing system atop the TLS-encrypted TCP bytestream. In TCPLS, we choose to implement multiplexing with TCPLS records. We dedicate a TCPLS record type to TCPLS stream data. Each stream has a separate cryptographic context allowing concurrent encryption and decryption of data within the same session with per-stream buffers. A TCPLS stream consists of a sequence of TCPLS stream data records encrypted with their cryptographic contexts. Each stream is attached to one TCP connection.

One simple way to provide separate cryptographic contexts is to use multiple application-level keys. But this is known to degrade the security properties proportionally to the number of additional keys [17]. To overcome this, we propose an Initial Vector (IV) derivation technique that enables independent encryption/decryption for each stream without this security degradation. We add two goals to the design of this technique. First, the TLS 1.3 wire format must be preserved to avoid possible middleboxes interference. Second, each record of the TCPLS session must have a unique nonce in order to maintain the security properties of AEAD.

Fig. 2 illustrates how the AEAD IV is computed for a given TCPLS record of a TCPLS stream. First the left-most bits of the IV derived from the TLS handshake are summed with the 32-bit TCPLS Stream ID. Then the right-most bits are XORed with the 64-bit stream record sequence number. Each TCPLS stream has a separate record sequence number space. These two operations guarantee that every

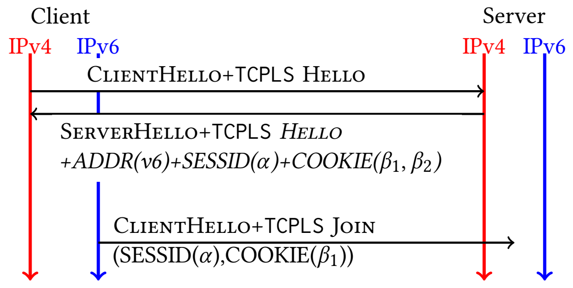
Client
Server
IPv4
IPv6
IPv4
IPv6
ClientHello+TCPLS Hello
ServerHello+TCPLS *Hello*
+ADDR(v6)+SESSID(α)+COOKIE(β_1, β_2)
ClientHello+TCPLS Join
(SESSID(α),COOKIE(β_1))

Figure 3: TCPLS supports joining additional TCP connections to a TCPLS session. The *SESSID* and *COOKIE* in the ServerHello are encrypted with the handshake key.

record of every TCPLS stream has a unique nonce. The Stream ID space is split between the client and the server. Stream ID 0 is equivalent to the cryptographic context derived directly from the handshake.

To avoid possible middleboxe interferences, in addition to the record sequence number, the TCPLS Stream ID is kept implicit as well. When an endpoint receives a TCPLS record, it leverages the AEAD cipher to check the authentication tag repeatedly until the correct TCPLS Stream ID is found. It does not involve a complete decryption as all TLS 1.3 AEAD ciphers use an Encrypt-then-MAC construct [36, 84], and the tag verification is computationally light, especially on AES-GCM [66]. Reducing this cost could be achieved by adding explicit signaling when another stream is scheduled over the TCP connection[2].

From a security standpoint, each failed decryption is considered as a forgery attempt. However, the limits on confidentiality and integrity of AEAD ciphers force the attacker to commit to an important number of attempts before a successful forgery may be considered with a non-negligible probability [35, 62]. For instance, in the case of ChaCha20 + Poly1305, an adversary has to perform 2^{60} forgery attempts before succeeding with probability 2^{-33}. Also, note that in case the adversary succeeds in decrypting the packet, it will very likely lead to a plain payload non-conforming to the TCPLS protocol.

3.3.2 Connection Migration. TCPLS allows joining several TCP connections to an existing TCPLS session. We leverage this mechanism to implement new transport services such as connection migration. Compared to the subflow joining mechanism of MPTCP [39, 40, 82], our design is more secure. Indeed, MPTCP supports additional subflows by initially exchanging short keys inside cleartext TCP Options during the TCP handshake [28, 29]. These keys are then used to authenticate the association of subflows. An attacker having observed the key exchange during the TCP handshake can associate subflows to an existing MPTCP connection [6].

Joining TCP connections. TCPLS leverages TLS extensions to solve this problem in a more secure manner. Fig. 3 illustrates the TLS and TCPLS messages exchanged when a client connects to a server over IPv4 and later joins another connection over IPv6. First, the

[2]The worst case is when the sender schedules records of N streams over a TCP connection. Then, TCPLS may announce within the last record of the current stream the Stream ID of the following record.

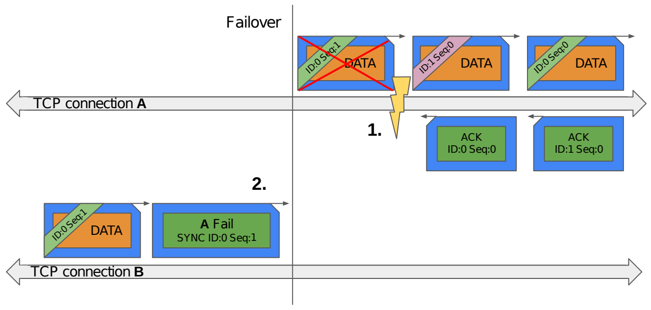

Figure 4: Failover resynchronizes and retransmits lost TCPLS records from a failed TCP connection to another.

client sends a ClientHello containing a TCPLS Hello. The server replies with a ServerHello containing three encrypted extensions. First, the server announces its IPv6 address ($ADDR(v6)$). Second, it associates one identifier α to the TCPLS session (*SESSID*(α)). Third, the server provides a list of TCPLS session cookies β_1, β_2 in the *COOKIE* extension. Each of these session cookies enables the client to join one additional TCP connection to the TCPLS session. Thus, by sending n cookies over a session, the server restricts the client to join up to n TCP connections. This prevents resource exhaustion attacks that are difficult to counter with MPTCP. The server can later send additional cookies and update its list of addresses.

To join a new TCP connection to the TCPLS session, for instance over IPv6, the client sends a ClientHello message containing the session identifier (SESSID(α) in Fig. 3) and one of the cookies provided by the server (COOKIE(β_1) in Fig. 3). The server uses the session identifier α to find the corresponding TCPLS session and checks the validity of the cookie. If the TCPLS session and this cookie are valid, the TCP connection is joined to the TCPLS session. The session identifier plays the same role as the MPTCP token, but is sent encrypted. The cookie provides stronger protection than MPTCP's HMAC with security keys exchanged in cleartext. The TCPLS cookies are sent encrypted by the server and can only be used once by the client.

Failover. When a TCP connection fails, e.g., due to middleboxes or network outages, TCPLS leverages its joining mechanism to recover the session over another TCP connection. This is particularly useful on multihomed devices that often move out of reach of a given access network, for instance a smartphone moving away from a Wi-Fi access point, but it also helps devices that suffer from middleboxes disrupting TCP connections, for instance a firewall introducing TCP RST on idle TCP connections. Fig. 4 illustrates how TCPLS reacts to such events during a transfer with two TCPLS streams. To achieve such a *break-before-make*, TCPLS sends acknowledgments for the received records of each stream, allowing the sender to remove them from its sending buffer. Looking at **1.** in Fig. 4, we can observe that three records were sent but only the first two were received and acknowledged due to a failure. Then, looking at **2.**, the sender notices that the TCP connection has failed and switches to the other one. It explicitly notifies the failure to the receiver and synchronizes the transmission sequence using a dedicated TCPLS record type (i.e., SYNC in Fig. 4). Then it retransmits the unacknowledged stream data records (i.e. the second record of stream 0 in Fig. 4). Explicit synchronization prevents a lost acknowledgment from desynchronizing the endpoints. Notifying the failure to the peer shortens its reaction time. Thanks to the per-stream cryptographic context, the ciphertext of lost stream data records can be retransmitted as is.

We present in Section 5.1 measurements quantifying the performance impact of adding TCPLS record-level acknowledgments. We also present in Section 5.3 an analysis of recovery speeds during different type of outages.

Application-triggered Connection Migration. TCPLS also enables the application to trigger a connection migration, e.g., based on application-level metrics qualifying its experience over the current path. For instance, a video streaming application could choose to move its connection to another access network whenever the obtained bandwidth does not satisfy the video bitrate requirements. TCPLS enables the exchange of meta-information to help this decision. For instance, an interactive application running on a smartphone could migrate from LTE to Wi-Fi when it senses an increase in delays due to bufferbloat for a given period of time, or when a mobile client detects its home Wi-Fi and the user's preference is to move its traffic to the Wi-Fi if detected. To achieve it, the client moves all its TCPLS streams to another TCP connection. First, it creates a new TCP connection and joins it to the TCPLS session. Then, it attaches new streams to the new connection and removes the old ones from the underperforming connection, effectively moving the application traffic to the new connection.

This second type of migration does not require application-level acknowledgments, but it cannot survive from one of the underlying connections' failure. During such a migration, data may be sent over two connections, which brings us to explain how multipath is designed.

3.3.3 *Multipath Capabilities.*

Like MPTCP, TCPLS allows an application to control different TCP connections that are used to exchange data. MPTCP was designed to be a drop-in replacement for TCP. A regular application that uses the socket API can use MPTCP without any modification. MPTCP uses a path manager to control the underlying TCP connections. Several path managers were proposed in the Linux kernel implementation [12, 39, 40]. However, few advanced applications have taken advantage of these advanced path managers. Apple's MPTCP stack was tuned for specific applications. For example, the Siri voice recognition application can use both cellular and Wi-Fi to optimize performance. Apple Music mainly uses MPTCP to perform seamless handovers. As applications have very different requirements, TCPLS exposes the underlying TCP connections to the application through its API. The applications use this API to implement their own policy to manage the underlying TCP connections.

Stream Steering. TCPLS enables the application to combine multiplexing with the ability to join several TCP connections to a TCPLS session. This allows the application to steer TCPLS streams over the different TCP connections. While Failover moves TCPLS streams from one connection to another when a TCP connection fails, stream steering allows the application to distribute at any time the streams over the different TCP connections of a TCPLS session. We already discussed a very simple use of stream steering when performing Application Connection Migration to move application data to a new connection.

Applications assign TCPLS streams to TCP connections through the TCPLS API. An interactive game could use different streams for chat messages and player's commands. Also, an HTTP server could choose the TCP connection for the stream of each response based on the content type of the response. Latency-critical objects could be sent over low-latency connections. By sending each application-level object in a separate stream, the application can benefit from multiple network paths while maintaining the ability to process each stream in a zero-copy manner, and with no head-of-line blocking. TCPLS enables the application to attach streams to separate TCP connections to avoid head-of-line blocking.

Coupling streams for aggregated bandwidth. Applications that benefit from the aggregated bandwidth of several network paths when transmitting a single application object can use TCPLS *coupled streams*. Coupled streams send data alongside a sequence number located at the end of the record. The sender can schedule TCPLS records of an application object over these streams and benefit from their aggregated bandwidth by sending data over the two streams using a strategy that fits the application usage. The receiving application reads the application object in order, as TCPLS handles the reordering of coupled streams decrypted records.

Dozens of packet schedulers were proposed for MPTCP. The default one is the RTT-based scheduler that favors the subflow with the lowest RTT [75]. Other schedulers include the redundant scheduler that sends data over both subflows [31], schedulers that help to minimize reordering on the receiver side [48, 59], schedulers specialized for mobile applications [22], ... The most flexible approach remains the application-defined MPTCP scheduler [32], but this solution has never been integrated in MPTCP implementations.

TCPLS takes a different viewpoint. It exposes the underlying TCP connections and the sender side TCPLS record scheduler to the application. This enables the application to actively decide the TCP connection that it will use to send each record. A remote terminal application running over TCPLS could send the screen updates over a high bandwidth but high latency connection and the keyboard input over the lower latency one. Using socket options such as `tcp_info`, an application can retrieve useful statistics about the performance of the underlying TCP connections (e.g. RTT, congestion window, ...). A more advanced application could also define TCPLS records to actively probe a connection, e.g. with an echo/request record to actively measure delays, or retrieve information from the remote host, e.g. by retrieving the remote host's `tcp_info` structure.

Coupled streams can also be used when performing Application Connection Migration to smoothly transition from one network path to another without impacting the application goodput.

3.4 Security and Information Leakage

Compared to MPTCP, the join mechanism of TCPLS allows cryptographically authenticated endpoints to add new TCP connections to an existing TCPLS session. It contrasts to MPTCP for which any observer of the initial handshake owns the required information to join the MPTCP connection. The multipath capabilities of TCPLS may also require stronger situational attacker models. One example would be preventing different IP networks from colluding and linking TCP connections belonging to a TCPLS session. The TCPLS handshake and the join mechanism can be enhanced to support this goal by leveraging encrypted tokens. One could then use a token for each connection join, effectively acting as both a session identifier and a cookie. Additional tokens could be generated and shared upon successful joins.

4 TCPLS PROTOTYPE

In this section, we describe our TCPLS implementation which provides the following benefits. (*i*, Sec. 4.1) Applications can use parallel streams with different cryptographic contexts and multiplex them over a TCP connection. (*ii*) TCP options can be sent through the secure TCPLS records, improving the extensibility of TCP (see [87]). (*iii*, Sec. 4.2) Applications can trigger Connection Migration and enable Failover. (*iv*, Sec. 4.3) Applications can leverage multipath capabilities such as stream steering and bandwidth aggregation. (*v*, Sec. 4.4) The server can securely send eBPF bytecode to the client to upgrade its TCP congestion control scheme or tune other TCP mechanisms [15, 99]. Through these examples, we demonstrate that our design enables modern transport services.

Our prototype is a fork of the `picotls` TLS 1.3 implementation [26] to which we added 9k lines of C code to implement TCPLS.

4.1 Multiplexing

We leverage the TCPLS records and the TCPLS stream cryptographic contexts to build our prototype with a zero-copy receive path. When a record is received, TCPLS first finds the corresponding cryptographic context, i.e., the corresponding TCPLS stream for which the derived IV leads to a successful authentication of the tag. This search is limited to the streams attached to the TCP connection of the received record. Our prototype tries first the last successful TCPLS stream. The receiver only has to find the right cryptographic context occasionally depending on the application, i.e., when the sender schedules another stream over the TCP connection.

At this stage, prior to full decryption, the TCPLS stream of the record is known. Our prototype can then locate the corresponding buffer to perform full decryption at the expected offset without any extra copy. Applications can benefit from large contiguous buffers to perform reads instead of receiving stream data on a per-packet basis, as often found in QUIC implementations.

4.2 Failover

We leverage the TCPLS records to securely exchange the TCP User Timeout option. This option enables endpoints to detect blackholed and failed TCP connections after a given time threshold. We use this as one trigger of the Failover. The sender configures this time threshold at the receiver side by sending this TCP option in a TCPLS record. The receiver can then notice when it stops receiving data and trigger the Failover. Receiving a TCP RST or FIN over a TCP connection for which TCPLS streams are attached does also trigger the Failover.

The stream-level acknowledgments required for the Failover to detect the lost TCPLS records can be enabled at the start of a TCPLS session. The default policy is to acknowledge every 16 received records, or when a stream has processed a given amount of bytes since the last acknowledgment.

Our prototype handles failover over IPv4 and IPv6 TCP connections, and by default, chooses different source and destination addresses than the failed TCP connection if possible.

Application running on more constrained devices can choose to disable Failover to gain in performance. They can also enable Failover during a TCPLS session by sending a message on the secure channel. We evaluate the performance impact of Failover in Sec. 5.3

4.3 Multipath

Stream steering. TCPLS exposes the TCP connections it manages to the application. This allows the application to directly distribute the TCPLS streams over the TCP connections at any point of the TCPLS session. A simple distribution is to move all TCPLS streams from one connection to another, as performed during Application-triggered Connection Migration. Our prototype enables these operations in a few API calls. More complex stream attachment policies can be implemented by the application to meet its requirements.

Bandwidth Aggregation. The application can create and join several TCP connections to the same TCPLS session. When coupled streams are attached to different TCP connections, TCPLS adds a sequence number encrypted in the TLS record payload. It is used to reorder the received records after decryption. Our prototype only supports coupling all streams together. The sending application implements the scheduling coupled streams over the different TCPLS connections. We expect to support other schedulers in the future and allow the application to select its preferred scheduler through the API, or even send it as eBPF bytecode over the session. The more TCPLS receives records in order, the more it can deliver them in a zero-copy fashion to the application. When a record is received out-of-sequence, its content is pushed on an efficient reordering heap. The performance of our multipath bandwidth aggregation is evaluated in Sec. 5.5.

4.4 eBPF Code Remote Attachment

Recent work on restructuring congestion control has proposed a generic architecture for congestion controllers [69]. Linux kernel developers have relied on eBPF to make the Linux TCP/IP stack easier to extend [15, 100]. Since Linux kernel version 5.6, an application can attach congestion control schemes entirely implemented in eBPF. A broader approach was proposed for QUIC in Pluginizing QUIC [23].

Our TCPLS prototype enables the server to attach a new eBPF congestion controller to the client over the TCPLS session. The type of eBPF code that can be attached could be easily extended to other points of the TCP execution path. The eBPF code is conveyed securely in a dedicated TCPLS record. When the code is larger than a single TLS record, it can be chunked in several records and sent using a TCPLS stream cryptographic context. This service illustrates how the flexibility of TCPLS record and streams can be leveraged to implement novel transport mechanisms.

4.5 TCPLS Session Establishment

Our prototype is fully compatible with TLS 1.3 0-RTT session resumption and TCP Fast Open option (TFO) [81]. By combining them, the TCPLS handshake can be sent together with the TCP SYN starting the three-way handshake. This provides a low-latency and secure connection establishment. It is not enabled by default in TCPLS, as TFO trades off some privacy [96]. More advanced techniques such as TCP Fast Open Privacy [96] could be integrated into TCPLS to solve this issue.

4.6 TCPLS API

The API used by applications to interact with a protocol plays an important role in leveraging all the protocol features. The most popular API to interact with the transport layer is the BSD socket API. Researchers and the IETF have explored other ways to expose a transport API [47, 77, 93, 104].

The TCPLS API builds upon good practices proposed by the outlined works. In this spirit, application-level developers are only required to configure a TCPLS context and register function callbacks. We design the TCPLS API such that it facilitates application development by offering a session-level interface based on asynchronous network events. The TCPLS API offers to application developers the opportunity to tune the transport protocol. They can then make the best choices for their applications.

As an example illustrating the flexibility of the TCPLS API, we consider a simple use case inspired by Happy Eyeballs [92]. This technique is used by web browsers when interacting with dual-stack servers. They start parallel TCP connections using IPv4 and IPv6 and then choose the one that offers the lowest latency. This avoids problems when one type of address is not supported on a network path but not the other, or when one results in lower latency [7].

Fig. 5 shows an example of our current API workflow. The API can handle explicit multipath techniques such as Happy Eyeball by chaining an optional call to `tcpls_connect()`, here with a time-out of 50ms, as shown in the Figure. TCPLS lets the application explicitly choose the addresses between which a TCP connection should be established by calling several times `tcpls_connect(src, dest, timeout)` in a TCPLS session. The application may configure callbacks to connection events occurring within TCPLS, such as connection establishments, stream attachments, multipath joins and the receipt of a TCP option to tune TCP.

4.7 TCPLS Fairness

Fairness is an important property of transport protocols guaranteeing an equal share of resources between users of the Internet. Our TCPLS prototype does not alter the congestion control algorithm of TCP. As such, a single connection of a TCPLS session is fair to a single TCP connection. However there are two situations in which this can be improved in future works.

First, when two or more TCP connections are used in a TCPLS session over network paths that share a common bottleneck, TCPLS can be unfair proportionally to the extent of this bottleneck. This problem is well known and solutions proposed for MPTCP such as coupled congestion control schemes [53, 78, 109] could be applied to TCPLS using eBPF.

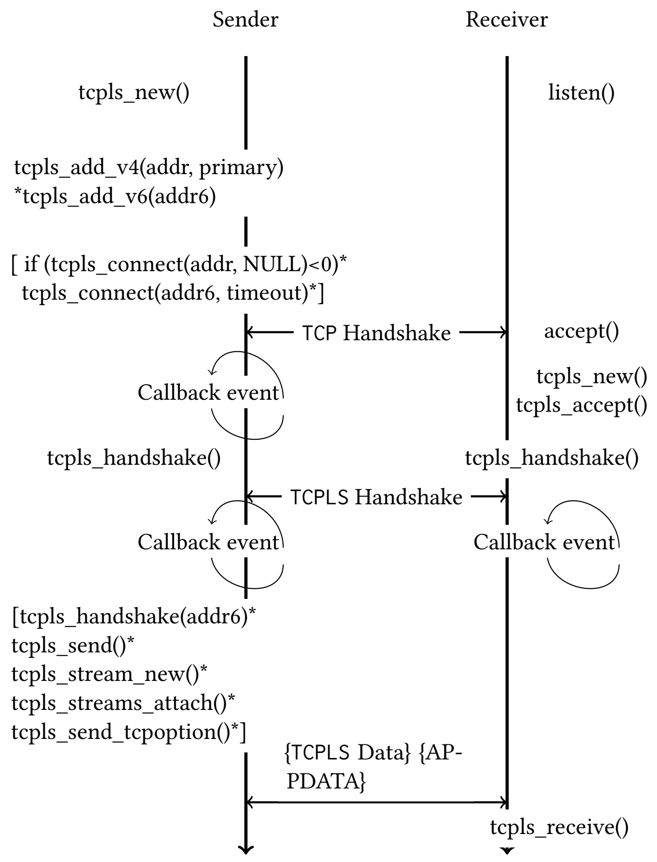

Figure 5: API Workflow example. * means optional call, [ ] means optional call flow, and { } means encrypted.

Second, when an application using TCPLS multiplexes several streams over a single connection, it has to compete with applications not using TCPLS that spread these streams over several connections, such as HTTP/1.1. This problem is also well known [68]. Recently, it led Google adapting mulTCP [20] when implementing gQUIC [56]. A similar approach could also be integrated in TCPLS with eBPF.

5 TCPLS EVALUATION

In this section, we evaluate TCPLS with several experiments. First, we evaluate the raw performance of our TCPLS prototype (Sec. 5.1). Then, we report how TCPLS interacts with middleboxes in a controlled environment in Sec. 5.2. We then emulate, using Mininet [38], more complex scenarios involving Failover (Sec. 5.3), Application Connection Migration (Sec. 5.4) and Bandwidth Aggregation (Sec. 5.5).

For the two first experiments of our TCPLS prototype (Sec. 5.1 and 5.2), we use the testbed depicted in Fig. 6. It consists of three servers equipped with Intel Xeon CPU E5-2630 2.40GHz, 16 GB RAM, running Debian with Linux 5.9 and 5.7 kernels. Two of these machines are used as, respectively, Client and Server, while the third one is used as a router or a middlebox, depending on the scenario. Each machine is equipped with an Intel XL710 2x40 Gbps

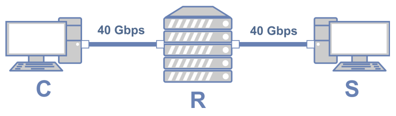

Figure 6: Performance Measurements Setup. C = Client. S = Server. R = Router/Middlebox.

NIC. For all measurements, with all implementations, we use a single thread on each machine to run the client and server.

In the emulated experiments of Sec. 5.3, 5.4 and 5.5, we use a Mininet network [38] composed of a client and a server, both dual-stacks. The IPv4 and IPv6 paths are completely disjoint in this emulated network, each offering 25 Mbps and 10 ms latency if not stated otherwise.

5.1 Raw Performance

In our first experiment, we demonstrate that TCPLS can be implemented at a low cost compared to TCP and TLS while offering better raw performance than several QUIC implementations. To this end, we use the physical testbed presented in Fig. 6 and run throughput measurements for large transfers with TCP/TLS, TCPLS and three QUIC implementations. The results are reported in Fig. 7 with the bandwidth in gigabits per second and packets per second obtained for each test. Each bar corresponds to the median measurement over 10 seconds of stable throughput. A preliminary test in our testbed shows that a single TCP connection can saturate the 40 Gbps link when setting the MTU to 9000 bytes. With a 1500-byte MTU, a single TCP connection reaches 22 Gbps. We also benchmarked the AES128-GCM-SHA256 cipher with in-memory 16,384-byte TLS records and observed that decryption in our client reaches 24.59 Gbps while encryption on our server reaches 13.62 Gbps.

TCP/TLS. To get a fair reference point, we first evaluate TLS over TCP using picotls with the AES128-GCM-SHA256 cipher at the commit starting our fork. We also manually configured the receiving and sending buffer size to match the ones we use internally in TCPLS. This modification fixes fragmentation and unnecessary copies of received TLS records that could occur and improves by ≈ 40% the throughput of picotls's measurement client. Our TCPLS prototype avoids these mistakes by design by offering to the application developers a transport API preventing them. Indeed, for example in TCPLS the question of *how much* the caller wants to read does not map to how much TCPLS should read from TCP due to the requirement of having a complete record to decrypt it. In this case, one may want to read more from TCP to prevent record fragmentation. So with this modification in picotls, we can observe that TCP/TLS reaches 10.3 Gbps with a 1500-byte MTU and 12.6 Gbps with a 9000-byte MTU. Both leverage TCP Segmentation Offload (TSO), which is a feature commonly enabled on high-speed NICs.

TCPLS. To benchmark TCPLS, we use a simple application performing large memory-to-memory transfers over a TCPLS session using a single stream, also configured to use the AES128-GCM-SHA256 cipher. We first compare TCPLS without the transport services presented in Sec. 3.3. We can observe that TCPLS reaches similar throughput than TLS/TCP over both MTU sizes. The small advantage of TCPLS at an MTU of 1500 bytes may be attributed by

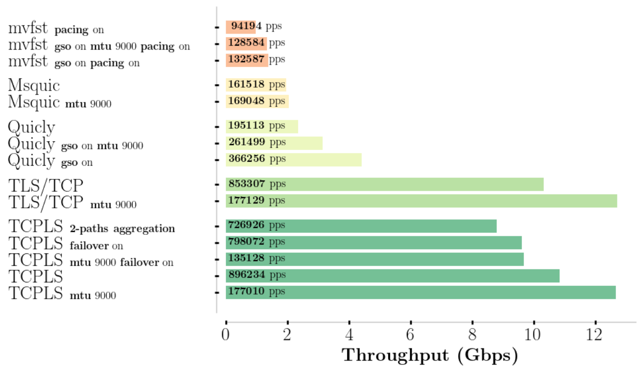

Figure 7: Throughput comparison between TCPLS, TCP/TLS, and three QUIC implementations. The TCPLS prototype is faster than several major QUIC implementations while offering a superset of transport services and similar flexibility.

implementation differences in the two benchmarks tools. TCPLS has thus a similar throughput than TCP/TLS when using the base protocol.

We then evaluate the cost of enabling Failover, which adds TCPLS-record-level acknowledgments. Our measurements show that Failover has a small impact on raw throughput as TCPLS reaches 9.66 Gbps with a 1500 bytes MTU. This is due to an increase in control records that are exchanged during the file transfer and additional buffer management within TCPLS. This impact could be reduced by lowering the number of record acknowledgments, which could further improve the throughput with an MTU of 9000 bytes. We leave finding the optimal acknowledgment frequency as future work.

Finally, we evaluate the cost of using coupled streams over several TCP connections. We configured TCPLS to use IPv4 and IPv6 as separate network paths and start a TCP connection on each path. The TCPLS server sends the records over the two TCP connections in a round-robin manner. The TCPLS client receives and reorders the records using an efficient heap. Our measurements show that coupling when two streams in our testbed, TCPLS reaches a throughput of 8.8 Gbps, i.e. less than 10% below Failover.

QUIC. We evaluate three representative QUIC implementations in our testbed: Facebook's `mvfst` [2, 52], Microsoft's `msquic` [1], and Fastly's `quicly` [4]. They are intended for production use and include throughput measurement applications. Furthermore, `mvfst` and `quicly` support Generic Segmentation Offload (GSO), which offloads UDP segmentation and checksum computation to the kernel and NICs. We configured the QUIC implementations to the most comparable setting as TLS/TCP and TCPLS. That is, we changed the cipher and set the CUBIC congestion controller when available.

The results obtained show that TCPLS compares favorably with the tested QUIC implementations. The fastest QUIC implementation is `quicly`, with GSO and a 1500-byte MTU, it reaches 4.4 Gbps. TCPLS with TSO is twice faster with the same CPU usage and similar receiving buffer sizes. Surprisingly, `quicly`'s performance decreases with jumbo frames but is still faster than without GSO. `msquic` reaches 1.96 Gbps while `mvfst` was slower despite GSO. The lower performance of QUIC implementations can be explained by several factors originating from the Linux UDP interface and QUIC design. (*i*) Early QUIC implementations use `sendmsg`/`recvmsg`, exchanging one packet at a time with the kernel. (*ii*) GSO is often not supported by the NIC and implemented in the kernel, unlike TSO. (*iii*) Pacing is implemented in userspace. (*iv*) Acknowledgments are handled in userspace, increasing further the number of context switches. (*v*) QUIC packets are smaller units than TLS records for encryption and decryption. While there are ongoing works to improve *i*, *ii*, and *iii* [3], it is likely that performance parity will require more offload capabilities, which could lessen the flexibility of QUIC. From a security standpoint, assuming NICs are opaque components of the network and not part of the user stack, it would be less compliant to the end users threat model.

Further works should also evaluate multiple TCPLS and QUIC sessions at the same time. In addition, the effect of multiple parallel streams for both protocol stacks would also be interesting to compare in a fairness evaluation over realistic Internet-like experimental conditions.

5.2 Middlebox Interference

We have tested TCPLS against different opensource and commercial stateful firewalls and proxy implementations (i.e., pfSense, IPFire, Cisco ASAv, mitmproxy) and found no unexpected interference. Stateful filtering and stateful packet inspection policies did not impact the TCPLS handshake and transparent TLS proxy successfully triggered TCPLS fallback to TLS/TCP. Still, the security appliances that block TLS 1.3 or some of its features [13, 57, 83] would also block TCPLS.

When faced with middleboxes that modify TLS 1.3 [13, 83], only one type of TCPLS messages can be impacted. The client TLS extensions, that are integrity-protected but not confidential, can be modified by these middleboxes. They contain messages such as TCPLS Hello, TCPLS Join, *SESSID*, and *COOKIE* presented in Sec. 3.3. Other messages conveyed in encrypted TLS records cannot be distinguished or tampered with. When a client opens a TCPLS session through a TLS terminating proxy not supporting TCPLS, it replies with a ServerHello message omitting the TCPLS Hello extension. Then, the client can implicitly fall back to TLS and continue with the handshake.

Some legacy TLS server implementations do not implement the TLS specification properly and might abort connections when receiving unknown TLS extensions. Similar behavior has been observed in overly restrictive stateful firewalls. To ensure connectivity in the presence of such policies, TCPLS implements an explicit fallback mechanism. If a client receives a TCP RST in response to the TCPLS ClientHello or no response, it tries negotiating a regular TLS connection after a timeout. In the same manner, a TCPLS Join extension might be blocked on a path when joining connections. In this case, the subflow attachment is canceled, and the application is notified to react appropriately, e.g., to cancel the migration.

A more complete evaluation of middlebox interference would require measurements in various operational networks that include such devices [46, 72, 83]. This is outside the scope of this paper and left for further work.

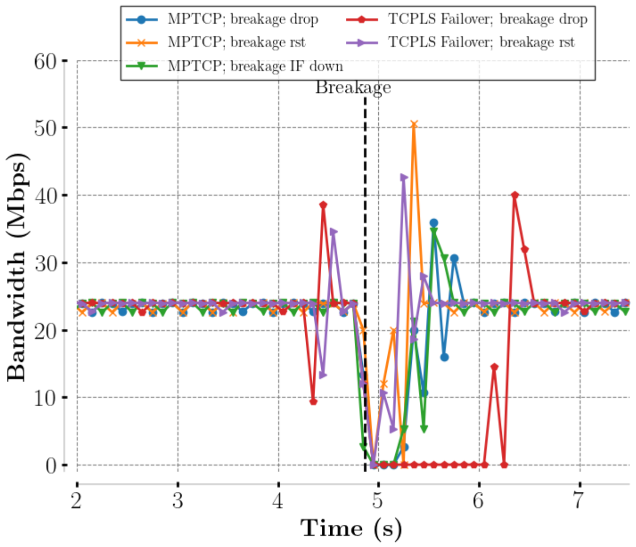

Figure 8: Recovery delays of TCPLS compared to MPTCP during a single outage.

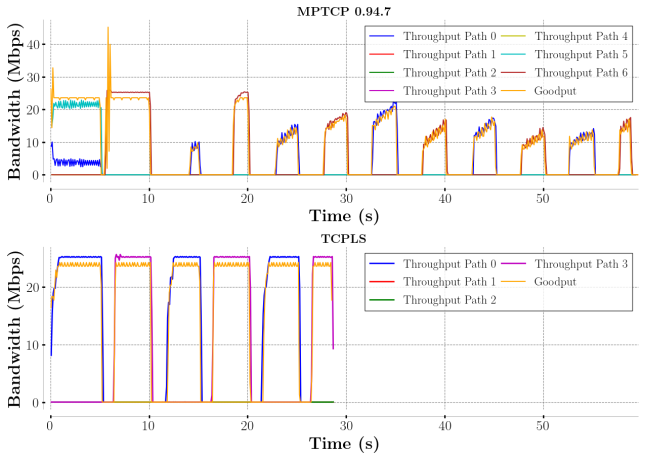

Figure 9: MPTCP has difficulties to react to successive network outages during a 60MB file download. TCPLS reacts quickly to such outages and completes the file transfer faster.

5.3 Failover

Failover is designed to provide resilience to TCPLS connection failures. Network outages may happen for several reasons, e.g., middlebox interference or mobile clients losing one access network (LTE or Wi-Fi). We first discuss and compare the recovery time for different types of outages during a file transfer for TCPLS and MPTCP[3]. We configure the client and the server with two interfaces, with one set to the backup mode in the case of MPTCP, i.e., no subflows are opened on the backup interface unless an issue is detected on the first interface. We consider two types of outages: a middlebox blackholing all the traffic and the reception of a spurious RST. Fig. 8 compares the goodput achieved by MPTCP and TCPLS over time when encountering these events.

We can observe that upon reception of a TCP RST, both TCPLS and MPTCP react fast. This is an explicit signal they both act on quickly and resume the transfer. However, a network outage is more difficult to detect. Both stacks rely on timers to decide to switch paths. In the case of TCPLS, we configure a timer using the TCP User Timeout option, which is set to 250 *ms* in our experiment. This value can also be chosen by the application according to specific use cases. When the timer expires, the client creates a connection over the other path and joins it to the session. The server then replays the unacknowledged records. Once these steps have succeeded, the transfer can continue. We can observe that this process takes ≈ 1 second to recover from this single outage in our experiment.

We now extend our first experiment by adding periodic outages during the transfer in a 4-path network topology. MPTCP is a more mature multipath transport implementation that is maintained and used in production. Fig. 9 compares how MPTCP and TCPLS recover when three paths out of four are blackholed every five seconds. We rotate the working path so that each implementation has first to find it before recovering the transfer. We observe that MPTCP's default path manager performs well on the first failure. For the next ones, it requires several seconds to recover the right path. We also tried injecting TCP RST instead of blackholing paths and MPTCP indefinitely stalled after the second RST. However, TCPLS finds the right path quicker and recovers the session in a short time similarly to the drop and RST outage studied in Fig. 8. Moreover, since those connections are fresh, they can quickly use the available bandwidth.

[3]Existing MPQUIC implementations [21, 23] are not evaluated as detecting and reacting to network failures were not part of their prototypes.

5.4 Application-Level Connection Migration

When there are multiple network paths, connection migration can be used by applications to adapt to the changing network conditions based on application metrics.

Applications can migrate their traffic from one network path to another using a few TCPLS API calls. Fig. 10 shows the goodput obtained when performing such a migration during a file transfer with TCPLS. We do not compare TCPLS to MPTCP and QUIC as none of their implementations offer this feature directly to applications. In this experiment, we reuse the Mininet topology and set a bandwidth of 30 Mbps on each path, a 40 ms round-trip-time for the IPv4 link, and 80 ms for the IPv6 link. Our test application downloads a 60 MiB file and migrates once from the IPv4 to the IPv6 path and once again back to the IPv4 path.

Fig. 10 reports the goodput obtained during the experiment. We can observe peaks during the migration windows marked with vertical black bars. During this window, the client uses coupled streams to transition smoothly from one TCP connection to the other, i.e., the first TCP connection finishes transmitting the queued TCPLS record while the second transmits the following records. TCPLS reorders the records and delivers the data to the client. The goodput increase corresponds to the additional bandwidth of the TCP connection over the new path during the migration time window. The application can thus trigger a connection migration and sustain its bandwidth during the process.

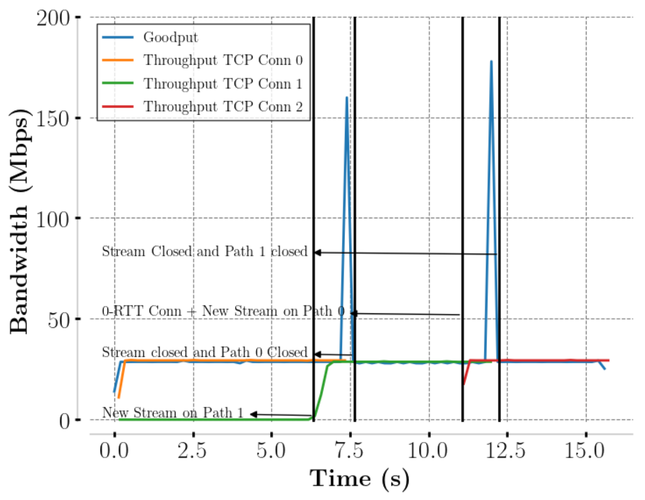

Figure 10: Application-level connection migration during a 60 MiB file download. TCPLS temporarily aggregates the two network paths during such a migration.

5.5 Bandwidth Aggregation

We compare the TCPLS bandwidth aggregation capability to the state-of-the-art MPTCP. Our experiment consists of transferring a 60 MiB file over a single path and enabling the second one after 5 seconds. MPTCP automatically detects the new path once we enable the Client's second network interface. For TCPLS, managing paths is different: the application can use the API to add local or peer-related addresses at any point of the session. In this experiment, we send this information after 5 seconds, create a new TCP connection to the peer, and attach a new stream to it. The two streams are coupled together. Our results appear in Fig. 11. Both protocols aggregate the two paths and reach a goodput of ≈ 50 Mbps.

We identify two differences. First, for MPTCP, there is a delay before it becomes fully utilized. This delay is the time required by the Linux kernel to configure the new network interface IP address, add the required routes, and finally inform MPTCP [74]. Second, TCPLS's aggregated goodput seems less stable than MPTCP. We explain this discrepancy by the difference of chunk size manipulated by the reordering algorithms in our experiment: MPTCP reorders packets with a payload of 1,460 bytes, while TCPLS reorders records with the maximum payload size of 16,384 bytes. As concurrent network paths introduce reordering, which is reordered back by both MPTCP and TCPLS, a larger chunk size leads to larger goodput irregularities.

Running the experiment with a record size of 1,500 bytes smooths out the irregularity at a slightly higher CPU cost since encryption and decryption are performed more often. Appendix A shows the results of the same experiment but using a TLS record size of 1,500 bytes instead. With these differences explained, we can conclude that TCPLS offers a bandwidth aggregation service similar to the one offered by state-of-the-art MPTCP.

5.6 Dynamically Extending TCPLS

To illustrate how TCPLS can be dynamically extended, we show how a server can send an eBPF congestion controller to its client

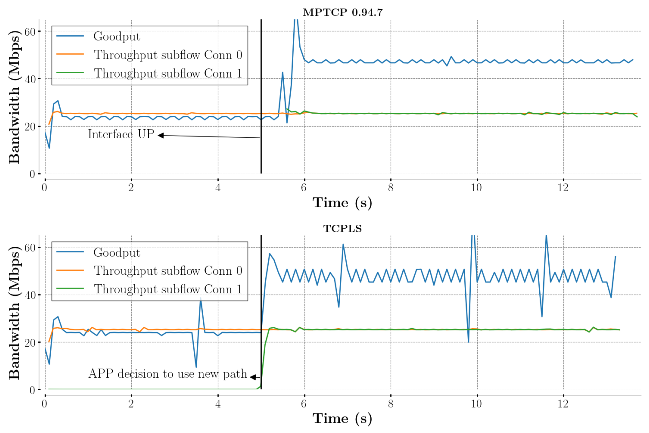

Figure 11: Bandwidth aggregation comparison between MPTCP (top plot) and TCPLS (bottom plot) with a record payload size of 16,384 bytes.

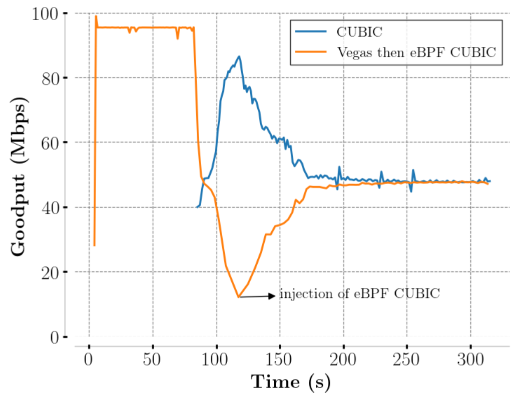

Figure 12: TCPLS hosts can exchange eBPF congestion controllers and enable them during a TCPLS session.

that attaches it in the middle of a TCPLS session. Fig. 12 reports the goodput obtained by two TCPLS sessions during a file upload on a Mininet network with a 100 Mbps and 60 ms RTT link. The orange curve is the first TCPLS session that starts with the Vegas congestion controller [16]. It rapidly reaches the full capacity of the link. Then, a second TCPLS session, depicted in blue, starts with the CUBIC congestion controller [85]. This quickly results in an unfair distribution of the bandwidth. The server then sends an eBPF bytecode implementing the CUBIC congestion controller to the first TCPLS session which attaches it for the current connection only. The bandwidth distribution becomes fair. We performed the same experiment using different delays, varying from 10 ms to 100 ms and obtained similar results.

6 RELATED WORK

By combining TCP and TLS, TCPLS builds upon two of the most important Internet protocols. Given their importance within the research community and the IETF, we restrict our discussion to close related works. Readers may refer to survey papers for additional context information [58, 76, 79].

Regarding its transport features, TCPLS' stream abstraction has similarities to the Structured Streams Abstraction [30], in which each stream does not need 3-way handshaking, is independent if attached to a different TCP connection and can proceed in parallel without head-of-line blocking if the streams are not coupled. However, TCPLS pushes further the abstraction, involving benefits leveraged from multiple connections.

TCPLS uses TLS records to encode data and control information. Researchers have also explored the idea of encoding control information in the TCP bytestream in different protocols. During the initial discussions for Multipath TCP, Multi-Connection TCP (MCTCP) [91] was proposed as an alternative that encodes control information in the bytestream. The MPTCP Working Group did not adopt this solution because it notably feared of possible problems with middleboxes. Multipath TCP [29, 82] uses TCP Options to encode control information and use different paths. TCPLS uses TLS records for this purpose and prevents middlebox tampering. Since TLS records are encrypted, their integrity is protected and content is obfuscated, so middleboxes cannot interfere with the control information that TCPLS exchanges. In contrast, with MPTCP, a middlebox that modifies the payload, such as a transparent TCP proxy or an application level gateway running on a NAT [44], can disrupt the protocol. Minion [71] also encodes control information in the TCP payload but to support unreliable data.

Given its security features, TCPLS must be compared with tcpcrypt [10, 11] which predates TLS 1.3. It uses TCP options to support opportunistic encryption but is not secure against an active network attacker. TCPLS extends TLS while retaining its security properties, and supporting new features. It is also compatible with middleboxes, as it leaves the TCP wire format untouched.

MPTCP [28, 82] supports several coupled congestion control schemes [53, 78, 109] that preserve fairness when different paths share the same bottleneck. A similar solution could be applied to TCPLS by leveraging eBPF to access and modify the congestion controller state. We leave this engineering effort as future work. The initial idea of coupling MPTCP and TLS was proposed in an Internet draft [73] that was not adopted by the IETF. MPTCPSec [51] adds security capabilities to MPTCP but comes with a large performance penalty.

Another important related work is QUIC. QUIC version 1 [50] also supports connection migration, but to our knowledge current implementations do not allow applications to trigger it. Multipath extensions [21, 60, 61, 105] to QUIC have been discussed but not yet adopted within the IETF. Finally, PQUIC [23] proposed to convey eBPF code over QUIC connections to deploy new protocol features. The same benefits are also being considered for other distributed systems, such as BGP [108] and anonymous overlay networks [88]. This goes beyond the exchange of a congestion controller in TCPLS.

Finally, several solutions have been proposed to provide multipath capabilities in the application layer. Examples include MP-H2 [70] that extends HTTP/2, MP-DASH [37] for video streaming or mHTTP [54] that extends HTTP.

TCPLS also share similarities with TLS FOP [96], which solves the privacy issue with TFO by using TLS. However, TCPLS is more generic, and its extensibility enables more advanced concerns and transport services.

7 CONCLUSION

TCP and TLS were designed as independent protocols, but they are very often used together. In this paper, we have shown that this is possible to design and implement the TCPLS protocol that is fast, flexible, and secure by closely coupling TCP and TLS.

TCPLS inherits the security features of TLS 1.3 and all the reliability and congestion control techniques that have been added to TCP during the last decades. More specifically, TCPLS extends the TLS 1.3 handshake and the record layer to create a secure control channel between the two communicating hosts. The messages exchanged over this channel are placed inside TLS records that are encrypted and authenticated and also hidden from middleboxes. TCPLS leverages these control messages to support fast failovers, smooth migration of the TCPLS session from one path to another and to provide bandwidth aggregation under full control of the application through an API. TCPLS can also use the secure channel to extend TCP with new options and even use the TCPLS session to exchange a different congestion control scheme that is then used for this session.

Thanks to TCPLS's design, our TCPLS implementation provides a flexible API that allows the applications to perform zero-copy data transfers, and easily manipulate the different features presented in this research work. The performance evaluation shows that our prototype is more than twice as fast as currently available QUIC implementations while supporting additional features such as bandwidth aggregation and stream steering.

TCPLS is both simple and powerful. Simple because it can be implemented inside existing TLS libraries without any kernel change in contrast to MPTCP. TCPLS is wire-compatible with the existing TCP middleboxes and can thus be used in all environments where TLS 1.3 is used over TCP. Given the performance benefits of TCPLS, the flexibility it offers and the features that it already provides, we believe that it can be a powerful contender to QUIC for modern services, including HTTP.

SOFTWARE ARTEFACTS

Our TCPLS prototype [86] is a fork of the picotls TLS 1.3 implementation. It contains about 9k additional lines of C code and supports the features described in the paper. In addition, it also includes support for QLOG/QVIS [64] similarly to several QUIC implementations. Instructions to reproduce our results are available [4].

ACKNOWLEDGMENTS

We thanks the anonymous reviewers for their helpful feedback. We thank Mathieu Jadin for his helpful guidance with IPMininet, Quentin De Coninck for his neat and available open-source tools, and Gaëtan Cassiers for helpful discussions. This research is supported by the Walloon Region through the "Programme de recherche d'intérêt général WALINNOV - MQUIC project (convention number 1810018)" and European Union through the NGI Pointer programme for the TCPLS project (Horizon 2020 Framework Programme, Grant agreement number 871528).

[4]https://github.com/frochet/tcpls_conext21

REFERENCES

[1] [n. d.]. Msquic implementation. https://github.com/microsoft/msquic. ([n. d.]). Accessed: Jan 2021.

[2] [n. d.]. mvfast implementation. https://github.com/facebookincubator/mvfst. ([n. d.]). Accessed: Jan 2021.

[3] [n. d.]. [PATCH RFC net-next 0/6] multi release pacing for UDP GSO. https://lwn.net/ml/netdev/20200609140934.110785-1-willemdebruijn.kernel@gmail.com/. ([n. d.]).

[4] [n. d.]. Quicly implementation. https://github.com/h2o/quicly. ([n. d.]). Accessed: Jan 2021.

[5] B. Anderson and D. McGrew. 2019. TLS Beyond the Browser: Combining End Host and Network Data to Understand Application Behavior. In *Proc. ACM Internet Measurement Conference (IMC)*.

[6] M. Bagnulo. 2011. *Threat Analysis for TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 6181. Internet Engineering Task Force.

[7] V. Bajpai and J. Schönwälder. 2019. A longitudinal view of dual-stacked websites—failures, latency and happy eyeballs. *IEEE/ACM Transactions on Networking* 27, 2 (April 2019), 577–590.

[8] R. Barik, M. Welzl, G. Fairhurst, A. Elmokashfi, T. Dreibholz, and S. Gjessing. 2020. On the usability of transport protocols other than TCP: A home gateway and internet path traversal study. *Computer Networks (COMNET)* 173 (May 2020), 107211.

[9] M. Belshe, R. Peon, and M. Thomson. 2015. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. Internet Engineering Task Force.

[10] A. Bittau, D. Giffin, M. Handley, D. Mazieres, Q. Slack, and E. Smith. 2018. *Cryptographic Protection of TCP Streams (tcpcrypt)*. RFC 8548. Internet Engineering Task Force.

[11] A. Bittau, M. Hamburg, M. Handley, D. Mazieres, and D. Boneh. 2010. The case for ubiquitous transport-level encryption. In *Proc. USENIX conference on Security*.

[12] L. Boccassi, M. M. Fayed, and M. K. Marina. 2013. Binder: A System to Aggregate Multiple Internet Gateways in Community Networks. In *Proc. ACM MobiCom Workshop on Lowest Cost Denominator Networking for Universal Access (LCDNet)*.

[13] K. Bock, iyouport, Anonymous, L.-M. Merino, D. Fifield, A. Houmansadr, and D. Levin. 2020. Exposing and Circumventing China's Censorship of ESNI. (August 2020). https://gfw.report/blog/gfw_esni_blocking/en/.

[14] D. Borman, R. Braden, and V. Jacobson. 1992. *TCP Extensions for High Performance*. RFC 1323. Internet Engineering Task Force.

[15] L. Brakmo. 2017. TCP-BPF: Programmatically tuning TCP behavior through BPF. In *Proc. Technical Conference on Linux Networking (Netdev 2.2)*.

[16] L. Brakmo, S. O'Malley, and L. Peterson. 1994. TCP Vegas: New Techniques for Congestion Detection and Avoidance. *ACM SIGCOMM Computer Communication Review* 24, 4 (October 1994), 24–35.

[17] S. Chatterjee, A. Menezes, and P. Sarkar. 2011. Another look at tightness. In *Proc. International Workshop on Selected Areas in Cryptography (SAC)*.

[18] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. 2014. *TCP Fast Open*. RFC 7413. Internet Engineering Task Force.

[19] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. 1989. An analysis of TCP processing overhead. *IEEE Communications Magazine* 27, 6 (June 1989), 23–29.

[20] Jon Crowcroft and Philippe Oechslin. 1998. Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP. *ACM SIGCOMM Computer Communication Review* 28, 3 (1998), 53–69.

[21] Q. De Coninck and O. Bonaventure. 2017. Multipath QUIC: Design and evaluation. In *Proc. ACM CoNEXT*.

[22] Q. De Coninck and O. Bonaventure. 2018. Tuning multipath TCP for interactive applications on smartphones. In *Proc. IFIP Networking Conference*.

[23] Q. De Coninck, F. Michel, M. Piraux, F. Rochet, T. Given-Wilson, A. Legay, O. Pereira, and O. Bonaventure. 2019. Pluginizing QUIC. In *Proc. ACM SIGCOMM*.

[24] K. Edeline and B. Donnet. 2019. A Bottom-Up Investigation of the Transport-Layer Ossification. In *Proc. IFIP Network Traffic Measurement and Analysis Conference (TMA)*.

[25] K. Edeline and B. Donnet. 2020. Evaluating the Impact of Path Brokenness on TCP Options. In *Proc. ACM/IRTF Applied Networking Research Workshop (ANRW)*.

[26] Kazuho Oku et al. 2021. `picotls` – TLS 1.3 implementation in C. (2021). https://github.com/h2o/picotls

[27] A. Feldmann, O. Gasser, F. Lichtblau, E. Pujol, I. Poese, C. Dietzel, D. Wagner, M. Wichtlhuber, J. Tapiador, N. Vallina-Rodriguez, O. Hohfeld, and G. Smaragdakis. 2020. The Lockdown Effect: Implications of the COVID-19 Pandemic on Internet Traffic. In *Proc. ACM Internet Measurement Conference (IMC)*.

[28] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. 2013. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 6824. Internet Engineering Task Force.

[29] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, and C. Paasch. 2020. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 8684. Internet Engineering Task Force.

[30] B. Ford. 2007. Structured streams: a new transport abstraction. In *Proc. ACM SIGCOMM*.

[31] A. Frommgen, T. Erbshäußer, A. Buchmann, T. Zimmermann, and K. Wehrle. 2016. ReMP TCP: Low latency multipath TCP. In *Proc. IEEE International Conference on Communications (ICC)*.

[32] A. Frömmgen, A. Rizk, T. Erbshäußer, M. Weller, B. Koldehofe, A. Buchmann, and R. Steinmetz. 2017. A programming model for application-defined multipath TCP scheduling. In *Proc. ACM/IFIP/USENIX Middleware Conference (Middleware)*.

[33] F. Gont and L. Eggert. 2009. *TCP User Timeout Option*. RFC 5482. Internet Engineering Task Force.

[34] QUIC Working Group. 2021. Available QUIC Implementations. (January 2021). https://github.com/quicwg/base-drafts/wiki/Implementations.

[35] F Günther, M. Thomson, and C.A. Wood. 2020. *Usage Limits on AEAD Algorithms*. Internet Draft (Work in Progress) draft-irtf-cfrg-aead-limits-00. Internet Engineering Task Force.

[36] P. Gutmann. 2014. *Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*. RFC 7366. Internet Engineering Task Force.

[37] B. Han, F. Qian, L. Ji, and V. Gopalakrishnan. 2016. MP-DASH: Adaptive video streaming over preference-aware multipath. In *Proc. ACM CoNEXT*.

[38] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. 2012. Reproducible network experiments using container-based emulation. In *Proc. ACM CoNEXT*.

[39] B. Hesmans and O. Bonaventure. 2016. An enhanced socket API for Multipath TCP. In *Proc. ACM Applied Networking Research Network (ANRW)*.

[40] B. Hesmans, G. Detal, S. Barre, R. Bauduin, and O. Bonaventure. 2015. SMAPP: Towards smart Multipath TCP-enabled applications. In *Proc. ACM CoNEXT*.

[41] B. Hesmans, F. Duchene, C. Paasch, G. Detal, and O. Bonaventure. 2013. Are TCP Extensions Middlebox-Proof?. In *Proc. Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*.

[42] K. Hickman. 1995. *The SSL Protocol*. Internet Draft (Work in Progress) draft-hickman-netscape-ssl-00. Internet Engineering Task Force.

[43] Paul Hoffman. 2002. SMTP Service Extension for Secure SMTP over Transport Layer Security. (2002).

[44] M. Holdrege and P. Srisuresh. 2001. *Protocol Complications with the IP Network Address Translator*. RFC 3027. Internet Engineering Task Force.

[45] R. Holz, J. Amann, A. Razaghpanah, and N. Vallina-Rodriguez. 2019. *The Era of TLS 1.3: Measuring Deployment and Use with Active and Passive Methods*. cs.CR 1907.12762. arXiv.

[46] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. 2011. Is It Still Possible to Extend TCP?. In *Proc. ACM Internet Measurement Conference (IMC)*.

[47] T. Hruby, T. Crivat, H. Bos, and A. Tanenbaum. 2014. On Sockets and System Calls: Minimizing Context Switches for the Socket API. In *Proc. USENIX Conference on Timely Results in Operating Systems (TRIOS)*.

[48] P. Hurtig, K.-J. Grinnemo, A. Brunstrom, S. Ferlin, O. Alay, and N. Kuhn. 2019. Low-latency scheduling in MPTCP. *IEEE/ACM Transactions on Networking* 27, 1 (February 2019), 302–315.

[49] J. Iyengar, P. Amer, and R. Stewart. 2006. Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths. *'IEEE/ACM' Transactions on networking* 14, 5 (October 2006), 951–964.

[50] J. Iyengar and M. Thomson. 2021. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. Internet Engineering Task Force.

[51] M. Jadin, G. Tihon, O. Pereira, and O. Bonaventure. 2017. Securing Multipath TCP: Design & implementation. In *Proc. IEEE INFOCOM*.

[52] M. Joras and Y. Chi. 2020. How Facebook is bringing QUIC to billions. (October 2020). https://engineering.fb.com/2020/10/21/networking-traffic/how-facebook-is-bringing-quic-to-billions/.

[53] R. Khalili, N. Gast, M. Popovic, and J.-Y. Le Boudec. 2013. MPTCP is not Pareto-optimal: Performance issues and a possible solution. *IEEE/ACM Transactions On Networking* 21, 5 (October 2013), 1651–1665.

[54] J. Kim, Y.-C. Chen, R. Khalili, D. Towsley, and A. Feldmann. 2014. Multi-source multipath HTTP (mHTTP) a proposal. In *Proc. ACM SIGMETRICS*.

[55] E. Kohler, M. Handley, and S. Floyd. 2006. Designing DCCP: Congestion control without reliability. *ACM SIGCOMM Computer Communication Review* 36, 4 (August 2006), 27–38.

[56] 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. Chang, and Z. Shi. 2017. The QUIC transport protocol: Design and Internet-scale deployment. In *Proc. ACM SIGCOMM*.

[57] H. Lee, Z. Smith, J. Lim, G. Choi, S. Chun, T. Chung, and T. Kwon. 2019. maTLS: How to Make TLS middlebox-aware?. In *Proc. Network and Distributed Systems Security (NDSS)*.

[58] M. Li, A. Lukyanenko, Z. Ou, A. Ylä-Jääski, S. Tarkoma, M. Coudron, and S. Secci. 2016. Multipath transmission for the Internet: A survey. *IEEE Communications Surveys & Tutorials* 18, 4 (2016), 2887–2925.

[59] Y.-S. Lim, E. M. Nahum, D. Towsley, and R. J. Gibbens. 2017. ECF: An MPTCP path scheduler to manage heterogeneous paths. In *Proc. ACM CoNEXT*.

[60] Yanmei Liu, Yunfei Ma, Quentin De Coninck, Olivier Bonaventure, Christian Huitema, and Mirja Kuehlewind. 2021. *Multipath Extension for QUIC*. Internet-Draft draft-lmbdhk-quic-multipath-00. IETF Secretariat. https://www.ietf.org/archive/id/draft-lmbdhk-quic-multipath-00.txt https://www.ietf.org/archive/id/draft-lmbdhk-quic-multipath-00.txt.

[61] Y. Liu, Y. Ma, C. Huitema, Q. An, and Z. Li. 2020. *Multipath Extension for QUIC*. Internet Draft (Work in Progress) draft-liu-multipath-quic-02. Internet Engineering Task Force.

[62] A. Luykx and K. Paterson. 2017. Limits on authenticated encryption use in TLS. (August 2017). https://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf.

[63] R. Marx, J. Herbots, W. Lamotte, and P. Quax. 2020. Same Standards, Different Decisions: A Study of QUIC and HTTP/3 Implementation Diversity. In *Proc. ACM Workshop on the Evolution, Performance, and Interoperability of QUIC (EPIQ)*.

[64] R. Marx, M. Piraux, P. Quax, and W. Lamotte. 2020. Debugging QUIC and HTTP/3 with qlog and qvis. In *Proc. ACM Applied Networking Research Workshop (ANRW)*.

[65] M. Mathis, J. NB Mahdavi, S. Floyd, and A. Romanow. 1996. *TCP Selective Acknowledgment Options*. RFC 2018. Internet Engineering Task Force.

[66] D. McGrew and J. Viega. 2004. The Galois/counter mode of operation (GCM). *submission to NIST Modes of Operation Process* 20 (2004), 0278–0070.

[67] A. Medina, M. Allman, and S. Floyd. 2004. Measuring Interactions Between Transport Protocols and Middleboxes. In *Proc. ACM Internet Measurement Conference (IMC)*.

[68] Jiwon Min and Youngseok Lee. 2019. An Experimental View on Fairness between HTTP/1.1 and HTTP/2. In *2019 International Conference on Information Networking (ICOIN)*. IEEE, 399–401.

[69] A. Narayan, F. Cangialosi, D. Raghavan, P. Goyal, S. Narayana, R. Mittal, M. Alizadeh, and H. Balakrishnan. 2018. Restructuring endpoint congestion control. In *Proc. ACM SIGCOMM*.

[70] A. Nikravesh, Y. Guo, X. Zhu, F. Qian, and Z. Mao. 2019. MP-H2: a Client-only Multipath Solution for HTTP/2. In *Proc. ACM Conference on Mobile Computing and Networking (MobiCom)*.

[71] M. F. Nowlan, N. Tiwari, J. Iyengar, S. O. Amin, and B. Ford. 2012. Fitting Square Pegs Through Round Pipes: Unordered Delivery Wire-Compatible with TCP and TLS. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[72] M. O'Neill, S. Ruoti, K. Seamons, and D. Zappala. 2016. TLS proxies: Friend or foe?. In *Proc. ACM Internet Measurement Conference (IMC)*.

[73] C. Paasch and O. Bonaventure. 2012. *Securing the MultiPath TCP handshake with external keys*. Internet Draft (Work in Progress) draft-paasch-mptcp-ssl-00. Internet Engineering Task Force.

[74] C. Paasch, G. Detal, F. Duchene, C. Raiciu, and O. Bonaventure. 2012. Exploring mobile/WiFi handover with multipath TCP. In *Proc. ACM SIGCOMM Workshop on Cellular Networks: Operations, Challenges, and Future Design (CellNet)*.

[75] C. Paasch, S. Ferlin, O. Alay, and O. Bonaventure. 2014. Experimental Evaluation of Multipath TCP Schedulers. In *Proc. ACM SIGCOMM Workshop on Capacity Sharing Workshop (CSWS)*.

[76] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K.-J. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic, and S. Mangiante. 2016. De-ossifying the Internet transport layer: A survey and future perspectives. *IEEE Communications Surveys & Tutorials* 19, 1 (2016), 619–639.

[77] T. Pauly, B. Trammell, A. Brunstrom, G. Fairhurst, C. Perkins, P. Tiesel, and C. Wood. 2020. *An Architecture for Transport Services*. Internet Draft (Work in Progress) draft-ietf-taps-arch-07. Internet Engineering Task Force.

[78] Q. Peng, A. Walid, J. Hwang, and S. Low. 2016. Multipath TCP: Analysis, design, and implementation. *IEEE/ACM Transactions on Networking* 24, 1 (February 2016), 596–609.

[79] M. Polese, F. Chiariotti, E. Bonetto, F. Rigotto, A. Zanella, and M. Zorzi. 2019. A survey on recent advances in transport layer protocols. *IEEE Communications Surveys & Tutorials* 21, 4 (2019), 3584–3608.

[80] J. Postel. 1981. *Transmission Control Protocol*. RFC 793. Internet Engineering Task Force.

[81] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. 2011. TCP Fast Open. In *Proc. ACM CoNEXT*.

[82] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. 2012. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[83] R. Raman, A. Stoll, J. Dalek, A. Sarabi, R. Ramesh, W. Scott, and R. Ensafi. 2020. Measuring the deployment of network censorship filters at global scale. In *Proc. Network and Distributed Systems Security (NDSS)*.

[84] E. Rescorla. 2018. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Internet Engineering Task Force.

[85] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger. 2018. *CUBIC for Fast Long-Distance Networks*. RFC 8312. Internet Engineering Task Force.

[86] Florentin Rochet. 2021. picotcpls, a first TCPLS implementation. https://github.com/pluginized-protocols/picotcpls. (2021).

[87] F. Rochet, E. K. Assogba, and O. Bonaventure. 2020. TCPLS: Closely Integrating TCP and TLS. In *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*.

[88] F. Rochet, O. Bonaventure, and O. Pereira. 2019. Flexible Anonymous Network. In *Proc. Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETS)*. https://arxiv.org/abs/1906.11520.

[89] J. Roskind. 2013. QUIC, Quick UDP Internet Connections. https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/preview. (2013).

[90] R. Sanders and A. Weaver. 1990. The Xpress transfer protocol (XTP) – a tutorial. *ACM SIGCOMM Computer Communication Review* 20, 5 (October 1990), 67–80.

[91] M. Scharf. 2010. *Multi-Connection TCP (MCTCP) Transport*. Internet Draft (Work in Progress) draft-scharf-mptcp-mctcp-01. Internet Engineering Task Force.

[92] D. Schinazi and T. Pauly. 2017. *Happy Eyeballs Version 2: Better Connectivity Using Concurrency*. RFC 8305. Internet Engineering Task Force.

[93] P. Schmidt, T. Enghardt, R. Khalili, and A. Feldmann. 2013. Socket intents: Leveraging application awareness for multi-access connectivity. In *Proc. ACM CoNEXT*.

[94] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. 2012. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. ACM SIGCOMM*.

[95] R. Stewart. 2007. *Stream Control Transmission Protocol*. RFC 4960. Internet Engineering Task Force.

[96] E. Sy, T. Mueller, C. Burkert, H. Federrath, and M. Fischer. 2020. Enhanced performance and privacy for TLS over TCP fast open. *Proc. Privacy Enhancing Technologies* (April 2020).

[97] J. Touch. 2013. *Shared Use of Experimental TCP Options*. RFC 6994. Internet Engineering Task Force.

[98] J. Touch and W. Eddy. 2018. *TCP Extended Data Offset Option*. Internet Draft (Work in Progress) draft-ietf-tcpm-tcp-edo-10. Internet Engineering Task Force.

[99] V.-H. Tran and O. Bonaventure. 2019. Beyond socket options: making the Linux TCP stack truly extensible. In *Proc. IFIP Networking Conference*.

[100] V.-H. Tran and O. Bonaventure. 2020. Beyond socket options: Towards fully extensible Linux transport stacks. *Computer Communications (COMCOM)* 162 (October 2020), 118–138.

[101] M. Trevisan, D. Giordano, I. Drago, and A. S. Khatouni. 2021. *Measuring HTTP/3: Adoption and Performance*. cs.NI 2102.12358. arXiv.

[102] M. Trevisan, D. Giordano, I. Drago, M. Munafò, and M. Mellia. 2020. Five years at the edge: Watching Internet from the ISP network. *IEEE/ACM Transactions on Networking* 28, 2 (April 2020), 561–574.

[103] M. Trevisan, D. Giordano, I. Drago, M. M. Munafo, and M. Mellia. 2020. Five Years at the Edge: Watching Internet from the ISP Networks. *IEEE/ACM Transactions on Networking* 28, 2 (April 2020), 561–574.

[104] M. Tüxen, V. Yasevich, P. Lei, R. Stewart, and K. Poon. 2011. *Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)*. RFC 6458. Internet Engineering Task Force.

[105] T. Viernickel, A. Froemmgen, A. Rizk, B. Koldehofe, and R. Steinmetz. 2018. Multipath QUIC: A deployable multipath transport protocol. In *Proc. IEEE International Conference on Communications (ICC)*.

[106] Zhaoguang Wang, Zhiyun Qian, Qiang Xu, Zhuoqing Mao, and Ming Zhang. 2011. An untold story of middleboxes in cellular networks. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 374–385.

[107] Nicholas Weaver, Christian Kreibich, Martin Dam, and Vern Paxson. 2014. Here be web proxies. In *International Conference on Passive and Active Network Measurement*. Springer, 183–192.

[108] T. Wirtgen, Q. De Coninck, R. Bush, L. Vanbever, and O. Bonaventure. 2020. xBGP: When You Can't Wait for the IETF and Vendors. In *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*.

[109] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. 2011. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

A MULTIPATH AGGREGATION

Figure 13 shows the same experiment as the one provided in Section 5.5. However, we now configure TCPLS with a 1,500 bytes record size to demonstrate that the irregularities are mainly due to the payload chunk size that the reordering algorithm has to manipulate. The larger the records, the bigger the payload chunk size, and higher irregularities should be observed. We observe here a steady goodput with several large irregularities, which is a progress compared to Figure 11. However, large peaks remain, which are probably linked to the some remaining bugs in our current TCPLS prototype implementation.

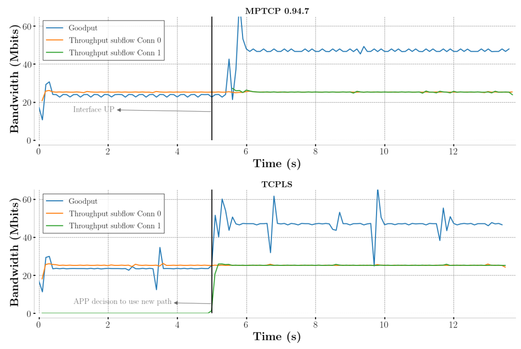

Figure 13: TCPLS path aggregation with 1500, bytes record size, compared to TLS over MPTCP with 16,384 bytes record size