UCLouvain

epl

École polytechnique de Louvain

TCPLS Evaluation & Extension

Author: **Aurélien Buchet**
Supervisor: **Olivier Bonaventure**
Readers: **Maxime Piraux, Florentin Rochet**
Academic year 2021–2022
Master [120] in Computer Science and Engineering

Acknowledgments

I would like to thank Prof. Olivier Bonaventure as well as Maxime Piraux and Tom Barbette for their support, pieces of advice and corrections throughout the year.

I would also like to express my gratitude to François Michel for his help with the NDK and to Florentin Rochet for accepting to be a reader of this thesis.

Finally, I would like to thank my friends and family who supported me during the good and the bad times. Special mention to Nikita Tyunyayev for all the hours of work together and all our discussions regarding our thesis during this year.

Abstract

TCPLS is a new protocol integrating TCP and TLS to provide transport services to applications while being resistant to middleboxes. In this thesis, we evaluate the first TCPLS prototype over two different aspects: the performance when using multiple streams and the portability to mobile devices. Our experiments highlighted the impact of the scheduling between streams on the performance and demonstrated that TCPLS could be used by an Android application to communicate with a server through cellular and WiFi networks. We also propose several extensions to the existing protocol to enable new services such as TCP information exchange, NAT detection and flow shaping. We integrated these extensions within the prototype to demonstrate their feasibility. Finally, we developed a protocol to create secured tunneled byte streams on top of TCPLS. We implemented a prototype of this protocol to evaluate its performance over different metrics: the goodput, the latency and the number of requests per second to demonstrate that it is usable. All the pieces of software that we used are publicly available.

Contents

1 State of the art **6**

1.1 Transport protocols 6

1.1.1 TCP 6

1.1.2 TLS 9

1.1.3 HTTP 12

1.1.4 QUIC 13

1.2 Middleboxes 13

1.2.1 Description of middleboxes 14

1.2.2 Deployment of middleboxes 14

2 TCPLS **16**

2.1 Principles 16

2.1.1 Session Establishment 16

2.1.2 TCPLS Records 17

2.1.3 TCPLS Transport Features 17

2.2 Evolution 21

2.2.1 Updated Transport Features 21

2.2.2 Record protection 22

2.2.3 Comparison 23

3 TCPLS evaluation **24**

3.1 Stream multiplexing 24

3.1.1 Decryption cost 24

3.1.2 Impact of the scheduler 25

3.1.3 Methodology 25

3.1.4 Results 27

3.2 Android support 29

3.2.1 Picotcpls Android library 29

3.2.2 TCPLS Mobile Application 30

3.2.3 Validation 30

4 TCPLS extensions **33**
4.1 TCP information exchange . . . 33
4.2 NAT detection . . . 35
4.2.1 Principles . . . 36
4.2.2 Implementation . . . 36
4.2.3 Packets format . . . 37
4.2.4 ICMP based middlebox detection . . . 38
4.2.5 Comparison with ICMP based detection . . . 38
4.2.6 Validation . . . 39
4.3 Flow shaping . . . 39
4.3.1 Principles . . . 40
4.3.2 Implementation . . . 40
4.3.3 Validation . . . 41

5 TCPLS use case: Tunneling Internet protocols inside TCPLS **43**
5.1 Opening a TCPLS Tunnel . . . 44
5.2 Messages format . . . 44
5.2.1 TCP Connect . . . 45
5.2.2 TCP Connect OK . . . 45
5.2.3 Error . . . 45
5.2.4 End . . . 46
5.3 Prototype . . . 46
5.4 Evaluation . . . 46
5.4.1 Experimental environment . . . 47
5.4.2 Goodput . . . 47
5.4.3 Connection latency . . . 48
5.4.4 Requests per second . . . 49

Introduction

Transport protocols have continuously evolved throughout the years by competing with each other. Adapting to the needs as networks grew bigger and performance became more and more critical. While TCP is the most used transport protocol nowadays, QUIC has gained in popularity since it was introduced. TCPLS is a new protocol combining both TCP and TLS to provide modern transport services. Using TLS allows overcoming the ossification of TCP due to the large deployment of middleboxes.

As TCPLS is a very recent protocol, there are lots of aspects that can be discussed. In this thesis, we analyze the performance of the first prototype when used with multiple streams as well as its portability to mobile devices. We also propose extensions to the protocol to provide new services such as the exchange of information regarding connections, detection of NATs, or limiting the usage of a connection. Finally, we developed a new protocol working on top of TCPLS to provide tunneling services leveraging the stream interface to create secured byte streams.

We divided our work into five chapters. The first chapter describes the principles and evolution of protocols that are used on the internet of today as well as some middleboxes and how they play an important role in modern networking. The second chapter focuses on TCPLS, introducing the key concepts and the evolution of the protocol from its first specification to a new version proposed at the IETF. The third chapter presents how we evaluate the TCPLS prototype regarding its multiplexing capabilities and portability to Android devices. The fourth chapter proposes extensions to the protocol and explains how we integrated and validated them with the TCPLS prototype. The fifth chapter shows a concrete use case of TCPLS using the protocol to create tunnels allowing hosts to exchange secured byte stream within untrusted networks while benefiting from the services of TCPLS. This chapter is followed by a conclusion summarizing our work and presenting the software artifacts that we built as well as discussing the future directions of this work.

Chapter 1

State of the art

1.1 Transport protocols

The goal of this section is to explain the evolution over the years of transport protocols and the principles of those that are used nowadays in order to be able to understand the objectives and challenges of TCPLS which be explained in more detail in the next chapter.

1.1.1 TCP

TCP (Transmission Control Protocol) [3] is the most widely used transport protocol today. It ensures that data is sent reliably between two endpoints without overloading either endpoint or the network by implementing congestion and receive buffer control. It just has to run above the Internet Protocol (IP) [4], which is supported by practically every network.

Principles

To initiate a TCP connection, TCP uses a three-way-handshake depicted in figure 1.1. A client needs to send a packet with the SYN flag set and a sequence number x. When this packet is received, the server replies with a packet that has the SYN flag and the ACK flag set. The acknowledgment number is set to x+1 and the sequence number to y. The connection is established after acknowledgment of the SYN packet so at this point the client has an open connection and can start sending data but the server waits for the reception of a packet with an acknowledgment number higher than y and inside its window. The values chosen for x and y should not be predictable in order to prevent attackers to establish a

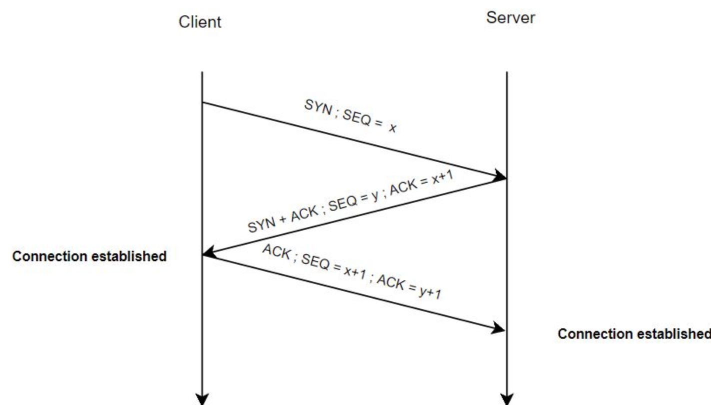


Figure 1.1: TCP handshake

connection by injecting packets into the network with a legitimate host's IP address.

In order to guarantee a reliable byte stream to the application, TCP uses re-transmittable packets. Each byte of data is identified by a 32 bits sequence number and needs to be acknowledged by a packet with an ACK flag set. When the ACK flag is set, the acknowledgment field contains the expected sequence number of the next data to receive. This indicates that all the data with a lower sequence number has been received. Receivers also indicate the current size of their receive window in the window field.

All TCP connections maintain a Transmission Control Block (TCB) containing the needed information to be able to deliver packets such as the next sequence number for both sides or the earliest not yet acknowledged sequence number. This TCB is updated whenever a packet is sent or received to reflect the current state of the connection.

The first retransmission strategy of TCP is based on timeouts. Whenever the retransmission timer expires, the first unacknowledged packet is retransmitted over the connection. The value of the retransmission timeout needs to be chosen carefully to avoid unnecessary retransmissions while detecting losses as fast as possible. Jacobson [1] proposed an algorithm to compute the retransmission timeout (rto) based on measures of the round-trip-time (rtt) by using two variables: the smoothed rtt (srtt) and the estimated variance of the rtt (rttvar). The rtt can be measured from the timestamp field using the RTTM option [8]. For the first measure of the rtt, the variables are computed as :

$$srtt = rtt \tag{1.1}$$
$$rttvar = rtt/2 \tag{1.2}$$
$$rto = srtt + 4 \times rttvar \tag{1.3}$$

Further updates are computed as follows :

$$srtt = (1 - \alpha) \times srtt + \alpha \times rtt \tag{1.4}$$
$$rttvar = (1 - \beta) \times rttvar + \beta \times |srtt - rtt| \tag{1.5}$$
$$rto = srtt + 4 \times rttvar \tag{1.6}$$

Where α and β are fixed parameters. The algorithm has been included in RFC 2988 [2] and the proposed values are $\alpha = \frac{1}{8}$; $\beta = \frac{1}{4}$. The advised initial value of the rto is three seconds and it should be rounded up to one second if it would be updated to less than a second. However, because latency has decreased since, current implementations use smaller values. For example, a recent version of the Linux kernel (5.17.9) uses a one-second initial timeout and 0.2 seconds as the minimum timeout value.

TCP Evolution

Over the years, the protocol had to be extended several times in order to keep up with new competing transport protocols. In 1990, Sanders et al. introduced the Xpress Transfer Protocol (XTP) [5] which was supposed to overcome the slowness of software-based TCP implementations by being designed for hardware. However, the TCP implementations were analyzed and optimized to make them support high speeds [6] resulting in the decline of XTP.

After these improvements to the speed of TCP, some major extensions were designed in order to enable scaling over gigabit link speed such as the TCP window scale extension allowing 32 bits representation of the receive window instead of the initial 16 bits one and timestamps extension for both round trip time measurements and protection against wrapped sequences [8].

Transport protocol researchers continued to develop alternatives to TCP and two new protocols were standardized by the IETF at the start of the 2000s: the Datagram Congestion Control Protocol (DCCP) [9] and the Stream Control Transmission Protocol (SCTP) [10]. DCCP aimed at providing congestion control for applications transferring fairly large amounts of data over unreliable datagrams but

is not widely used nowadays. SCTP was designed to overcome some limitations of TCP such as the lack of support for multihoming and its vulnerabilities to denial of service attacks. It was also easier to extend but even though SCTP offered an alternative to TCP, it was never widely deployed. We can explain this lack of popularity by the fact that applications that were already working with TCP would have had to change in order to integrate SCTP. Furthermore, many operators and enterprises use middleboxes that only support TCP or UDP traffic.

In parallel with the deployment of SCTP, TCP continued to be extended to adapt to the current needs including defenses against denial of service with TCP SYN flooding attacks [11]. Multipath TCP [12, 13] provides multihoming capabilities equivalent to SCTP as well as coupled congestion control [14].

1.1.2 TLS

The Transport Layer Security (TLS) is a commonly used protocol for encrypting and authenticating messages during communication between two peers. It was first introduced in the 1990s as the Secure Sockets Layer (SSL) [16] and has since then, evolved towards TLS 1.3 [17], the latest version. TLS is used to provide a layer of security to numerous application protocols, such as HTTP/2 for online web browsing and SMTP for mail services. TLS is most often used over TCP connections. However, it is compatible with any transport protocol that can provide an in order and reliable stream and has been recently included in QUIC [7]. One difference between QUIC with TLS and TLS over TCP is that QUIC cryptography is applied on QUIC packets while a TLS record could span over multiple TCP segments.

The original TLS 1.2 [18] handshake used a 4-way handshake illustrated in figure 1.2, records in red are encrypted. The client first sends a ClientHello message with a random nonce. The server replies with a ServerHello message containing its own random nonce as well as a Certificate message with the server's certificate or chain of certificates. Once the client has verified the certificate of the server, it uses the nonces as seeds to generate a secret that will be the base of the cryptographic keys. The client sends a ChangeCipherSpec indicating that it has all the cryptographic information and that it will begin the encrypted communication and follows by a Finished message containing a hash of all messages sent throughout the handshake. The server then decrypts the Finished message and checks that there was no modification in the handshake then, sends its own Finished message.

Combined with the TCP handshake, the TLS 1.2 connection establishment

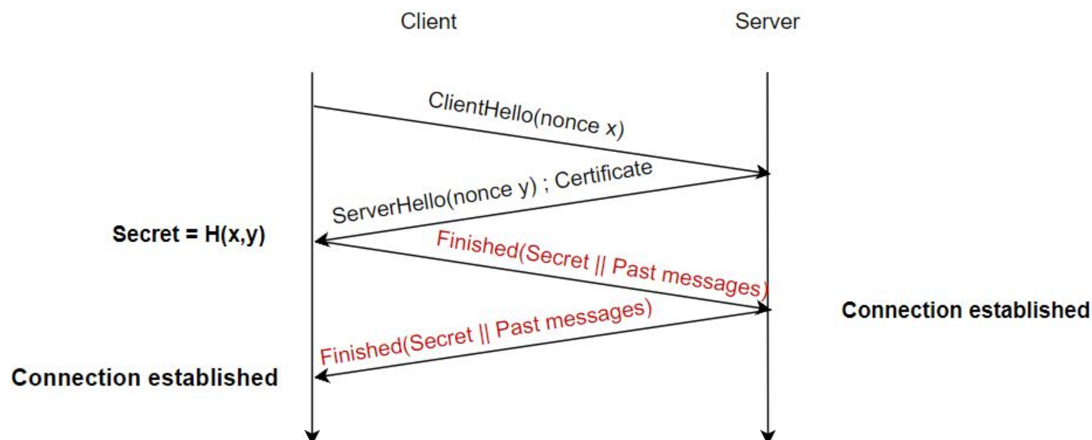
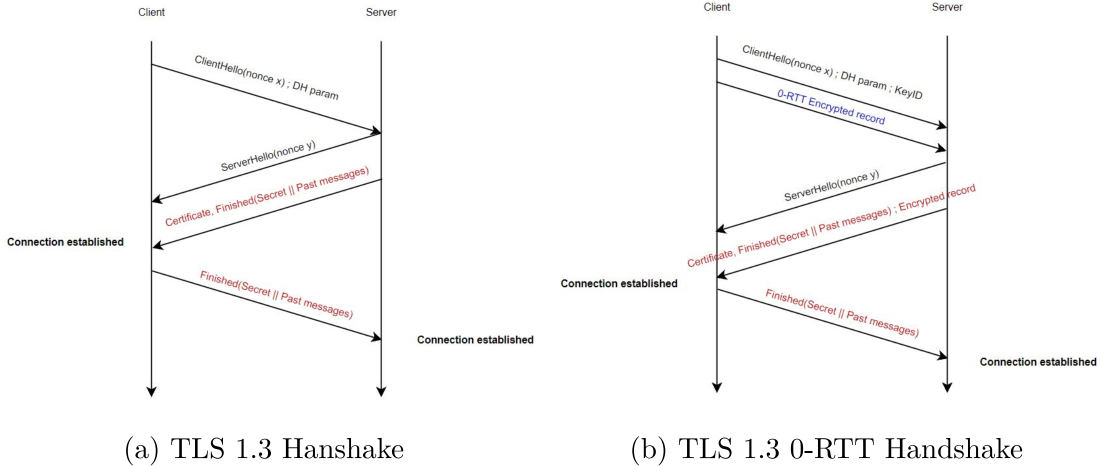


Figure 1.2: TLS 1.2 Handshake

required many RTTs and had a huge impact on latency. TLS 1.3 uses way fewer cipher suites and they all require a Diffie Hellman key exchange to ensure perfect forward secrecy (PFS) which guarantees that keys won't be compromised even if the private key of the server would be disclosed. This enables the client to already send the parameters for his preferred suite along with the ClientHello and have high a chance that the server supports them. If that is the case, the server can already derive the keys, send the ServerHello, and begins to send encrypted record. The server still needs to send the Certificate and Finished messages which must be verified by the client who can then also sends its own Finished message and sends encrypted data. This handshake is represented in figure 1.3a. The server and the client can also agree on a pre-shared key and an identifier to this key during a TLS session. These can later be used by the client to send encrypted records without having to wait for the reply of the server and upgrade later to the key of the current session. These mechanisms shown in figure 1.3b enable encrypted data transfer after only one RTT for the first session and 0 for successive ones.

After the keys have been derived from the secret exchanged during the handshake, all messages are encrypted and authenticated. TLS 1.3 guarantees privacy, authenticity and integrity of the data using Authenticate Encryption with Additional Data (AEAD) cipher suites. Each record contains four fields :

- The *Type* indicating whether the record corresponds to encrypted data, a handshake message, a change of cipher or an alert. In TLS 1.3, all messages use the AppData message type and store the true type encrypted at the end of the payload to avoid middlebox interference.



(a) TLS 1.3 Hanshake (b) TLS 1.3 0-RTT Handshake

Figure 1.3: TLS 1.3 1-RTT & 0-RTT Handshakes

- The *Protocol Version* indicating the version of TLS. In TLS 1.3, this value is always 0x303 corresponding to TLS 1.2. The actual version is negotiated during the handshake and can no longer change after the session has started.
- The *Length* Indicating the length of the record. The maximum size of a TLS record is 2^{14} bytes but can contain up to 256 bytes of additional data from the AEAD algorithm.
- The *Fragment* that contains the actual data transmitted

To be able to authenticate records, AEAD cipher suites us Message Authentication Codes (MAC) based on the plain text to be encrypted, the header of the record and an implicit sequence number maintained by both the client and the server but never explicitly sent. When the cipher text is decrypted, the MAC is computed and compared with the value in the record and if they don't match, an alert is sent stopping the session. It is also possible to compute the MAC with the cipher text instead of the plain text. This approach called encrypt-then-MAC was proposed in RFC7366 [19] and is considered more secure than the MAC-then-encrypt construction while enabling to check the authentication code before attempting to decrypt the record.

In parallel with the development of TLS, DTLS, a protocol providing equivalent security without ordering was designed for datagrams mainly used by UDP [20]. The first version, DTLS 1.0 [21] matched TLS 1.1 specifications. After TLS 1.2 was designed, the protocol evolved directly to DTLS 1.2 [22] to make the version numbers correspond. The latest version is DTLS 1.3 [23] and is based on TLS 1.3.

1.1.3 HTTP

The Hypertext Transport Protocol (HTTP) [24] was developed as part of the World Wide Web in the late eighties. As the name suggests, it is used to transfer hypertext documents but can also serve different purposes such as distributed object management using REST API.

The protocol is stateless and works with requests and responses. In the first versions of the protocol (HTTP/1.0 and HTTP/1.1), both the requests and the responses were text-based. A request is made of three different parts: the *method*, which indicates the type of request and the URI concerned as well as the version of the protocol, the *header* containing all parameters used for the request and for some request, a document can be attached. Responses are also divided into three parts: the *status-line* indicating how the request has been handled, a *header* with complementary information about the response and, in most cases, an attached document.

In the first versions of HTTP (HTTP/1.0), each request was associated with one specific TCP connection meaning that for web pages needing to load lots of documents, a high amount of TCP session establishment was needed which is highly inefficient. To remove this issue and to adapt to the fast growth of the web, the protocol quickly evolved towards HTTP/1.1 [25] which includes Keep-Alive options in the header to make connection persistent, request pipelining and more header options.

Over the last decade, the popularity of HTTP has continued to increase and it is now used by many interactive applications while still being the main protocol to transfer static documents. With the increase in the number of requests and the total volume of data needed to load web pages, HTTP/1.1 can struggle to provide good performances especially when low latency is required. One of the main problems is head of line blocking. As HTTP/1.1 supports pipelining, the client can send multiple requests without having to wait for older responses. This reduces the RTT but as requests are processed sequentially, if one takes more time because the object is bigger or because of a loss, the other ones become blocked.

To alleviate these issues, the IETF started working on an improved version of the protocol presented in 2014 as HTTP/2 [27] and standardized in 2015 [26]. The key ideas of this new version are that it is still fully compatible with version 1.1, it simply adds an upgrade option in the HTTP/1.1 header of the request to indicate to the server the support of HTTP/2. The format of the messages no longer relies on text messages but instead, binary frames. This makes them less understandable

for the developers but way easier to parse for computers. In addition, headers can be compressed using HPACK [28] to further reduce the size of the frames. Each frame is associated with a stream identifier that corresponds to a specific sequence of frames exchanged by a client and a server. A connection may support many different streams opened at the same time and schedule the frame of each stream to avoid head of line blocking when there are no losses. Streams can also have different priorities and be marked as dependent on other streams.

1.1.4 QUIC

In parallel with the development of HTTP/2, Google started working on a new protocol that could replace TCP and TLS from the HTTPS stack by providing secured in-order reliable streams of data that can be multiplexed. This protocol called QUIC (Quick UDP Internet Connections) [29] is built on top of UDP as many firewalls block all IP packets that do not contain a TCP or UDP packet and UDP is available for most users.

QUIC includes authentication and encryption directly inside the protocol. It Provides both the security of TLS 1.3 and the reliability of TCP. The Connection establishment includes the TLS 1.3 handshake giving the possibility to use the 0-RTT option.

Having the streams directly present at the transport layer reduces the problem of head of line blocking that could happen when using multiplexed HTTP/2 streams with TCP as TCP delivers all the data in order, a loss in one stream delays all of them while a loss in a QUIC stream only affect this particular stream.

The first QUIC implementations were made to be run from user-space in order to be easily updated and maintained as it does not rely on kernel updates like TCP. The drawback of running the protocol in user space is that it requires more context switches which lower performances at high speeds.

1.2 Middleboxes

Middleboxes are present in many enterprises networks [30] and in most modern cellular networks [31]. In this section, we define several kinds of middleboxes, the impact they can have on networks.

1.2.1 Description of middleboxes

Middleboxes are defined by the IETF as "any intermediary device performing functions other than the normal, standard functions of an IP router on the datagram path between a source host and destination host." [32]

This definition covers a large variety of behaviours from IP firewalls to HTTP caches but we will focus on the ones that directly impact the transport layer. Middleboxes can for example look into headers sent in clear such as the TCP one to detect options that are used and decide to disable some of them. This is one of the reasons why fields appearing in clear-text in the TLS records became ossified and the actual information is now sent encrypted. Other middleboxes such as NATs or TCP proxies go further by breaking the end-to-end principle of the protocol by opening implicit or explicit intermediates connections.

NATs are a specific kind of middlebox that maps a combination of an IP address and a port into another without the knowledge of the host. This implies that all packets to and from that address have the relevant fields translated on the fly including all checksums affected by the change (IP, TCP, UDP). They are particularly useful to overcome the shortage in IPv4 addresses by allowing multiple hosts in a local area network to connect to the Internet through a NAT using only one public IP address.

A proxy is an intermediate application between two endpoints that can evaluate requests and fulfill them on behalf of clients requesting its services. Proxies usually operate at the application layer to serve HTTP or DNS requests but they can also be configured to create transport tunnels.

1.2.2 Deployment of middleboxes

Middleboxes are now almost everywhere. In 2011, Wang et al. analyzed the deployment of middleboxes inside cellular networks [31]. They developed NetPiculet, a tool using a mobile application and a server outside the mobile network. NetPiculet generates and analyses traffic with TCP and raw sockets in order to identify specific NAT and Firewall policies.

They showed that most cellular carriers were using NATs and Firewalls to provide data service using limited IP address space and protect mobile devices from malicious attacks from the Internet. The configuration of these middleboxes directly impacts the performance of applications using cellular networks especially

when the developers are not aware of their presence. For example, applications using push-based messages with long-lived connections can be disturbed by NATs that are using small timeout values for inactive TCP connections because they would need to keep reopening the connections wasting resources.

Tracebox, a tool developed by Detal et al. [44] was deployed on PlanetLab [46], an open platform for planetary-scale network services, and observed paths towards 5,000 targets selected from the top Alexa websites. The experiment included 72 machines, each one having a shuffled version of the list. Analysis of the data gathered showed that many kinds of interference occurred. In addition to classical NATs and proxies behaviour, observed changes included variation in the TCP sequence number, the MSS and replacement of the Multipath TCP [12], MD5 [47], and Window Scale options [8] with NOP.

This shows that middleboxes are widely used in modern networking and they frequently alter packets that go through them modifying the behaviour of transport protocols. But by being aware of this, it is possible to detect these modifications to understand how middleboxes work and how to manage them.

Chapter 2

TCPLS

In this chapter, we present the principles of TCPLS as described in the first version by Rochet et al [33], and the evolution of the protocol towards a standardized version within the IETF [34]. We use TCPLSv1 to refer to the first version and TCPLSv2 to refer to the one presented at the IETF. As our main focus was on the first version, if no version is specified, we refer to TCPLSv1. The following chapters will present our work with the protocol.

2.1 Principles

TCPLS is a cross-layer protocol integrating TCP and TLS. It is designed to improve TCP by adding new services without directly modifying the protocol. The main goals of TCPLS are to

- Ease the deployment of current and upcoming TCP extensions
- Provide advanced transport features to applications
- Be a good challenger to QUIC with modern applications

In order to reach these goals, TCPLS leverages the extensibility of TLS 1.3 records to provide a channel that can be used to exchange control information without relying on TCP headers. Using TLS enables the messages to be encrypted and authenticated, reducing the risk of middlebox interference.

2.1.1 Session Establishment

The TCPLS session establishment is a TCP handshake followed by a TLS handshake with a TCPLS Hello extension within the initial ClientHello record. A TCPLS

server always indicates its support of TCPLS in its ServerHello record using TLS Encrypted Extensions. This way, if the client's TCPLS extension is blocked by a middlebox, it is still possible to enable TCPLS using encrypted control messages after the handshake. This makes sure that TCPLS cannot be distinguished from TLS from a middlebox point of view preventing any attempt to disable TCPLS without also blocking TLS.

TLS Encrypted Extensions can also be used to convey TCP options. These options are not restricted by the TCP header size and are hidden from middleboxes. Furthermore, some TCP extensions such as the TCP User Timeout option [36] should be sent reliably which is the case if they are in TLS records themselves contained in TCP packets.

2.1.2 TCPLS Records

TCPLS exchanges control information and application data inside specific TLS records by adding two new types of records: TCPLS Data and TCPLS Control. TCPLS Data records contain application data while TCPLS Control records are used for control information and have an additional type specifying the kind of information they transport. This type is a 32 bits value located at the end of the payload, just before the TLS record type. The formats of the different TCPLS records are represented in figure 2.1

2.1.3 TCPLS Transport Features

Applications can benefit from advanced transport features by leveraging TCPLS records to manage the underlying TCP connections. We present several of these mechanisms and how they are handled by TCPLS.

Stream Multiplexing

Streams are a used in most of the recent protocols like SCTP [10], HTTP/2 [26] and QUIC [29]. TCPLS Streams can be opened and closed by sending a TCPLS Control record with the Stream Attach or Stream Close type. TCPLS Data records are used to exchange stream data. To allow concurrent encryption and decryption over multiple streams in a secure manner, each stream must have its own cryptographic context based on the one derived from the handshake. The AEAD Nonce used by each TCPLS record is computed from the TLS Initial Vector (IV) in which the first 32 bits are added with the stream ID and the 64 last bits XORed with the stream sequence number maintained implicitly by each side. The derivation of the AEAD Nonce for a TCPLS record is represented in figure 2.2 The stream with ID

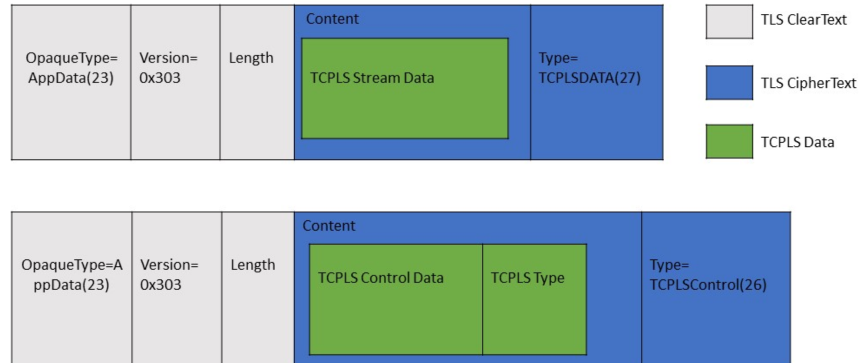


Figure 2.1: TCPLS Data and TCPLS Control format

0 corresponds to the context derived during the handshake and is used for control messages such as the ones used to open new streams.

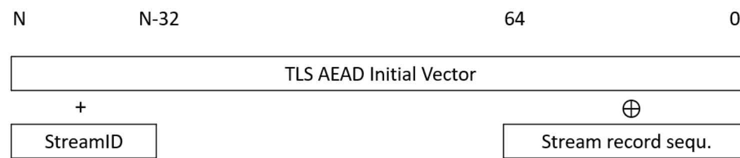


Figure 2.2: AEAD Nonce of TCPLSv1 Streams Records

The stream ID is not part of the record, it is kept implicit. In order to identify the stream to which the data belongs, the receiver has to verify the tag of the record for all streams attached to the TCP connection using its ID and the associated record sequence number until the correct context is found. More details about the decryption mechanisms and costs are discussed in section 3.1.

Connection Multiplexing

It is possible to have multiple TCP connections linked to one TCPLS session using TLS Extensions or TCPLS Control records. The ServerHello can contain different

Encrypted Extensions allowing the client to join the session :

- The SessionID Extension to identify the session with an encrypted 16 bytes value.
- The Multihoming V4/V6 Extensions to advertise the server's IP v4/v6 addresses.
- The Cookie Extension providing one or more 16 bytes session cookies.

The client can open new TCP connections using the addresses advertised by the server and join the TCPLS session by sending a TLS ClientHello with the TCPLS Join Extension containing the sessionID and a valid session cookie. The sessionID makes it possible for eavesdroppers to detect that multiple TCPLS Join are correlated. The server verifies that the values are correct and links the connection to the session. The number of cookies is controlled by the server and each cookie can only be used once to prevent resource exhaustion. Figure 2.3 illustrates how a multi-homed client can join a multi-homed server using these TLS Extensions.

TCPLS can use record-level acknowledgments, together with the join mechanism, to trigger a failover when one TCP connection fails. If the sender notices a failure in a TCP connection, it can switch to another one, explicitly indicating the failure and synchronizing the sequence numbers. This way, the peer can quickly react and the records can be retransmitted directly as the cryptographic context is stream-based and not connection-based. This reduces the latency but as the tag is exactly the same, eavesdroppers can detect that the two connections are correlated.

It is also possible to trigger a migration without failure by directly opening new streams on a new connection and closing streams attached to the old one.

Multipath

Combining connection and stream multiplexing allows applications to benefit from having multiple paths to join a server in several ways. TCPLS streams buffers can be used in two different modes.

The stream mode associates one buffer per stream. With this mode, it is possible to map streams to several TCP connections. Head of line blocking could still happen between streams belonging to the same connection but streams using separate connections should not be delayed by each other. When loading a web page, latency-critical streams like the HTML document can be mapped to a different connection than streams used for images.

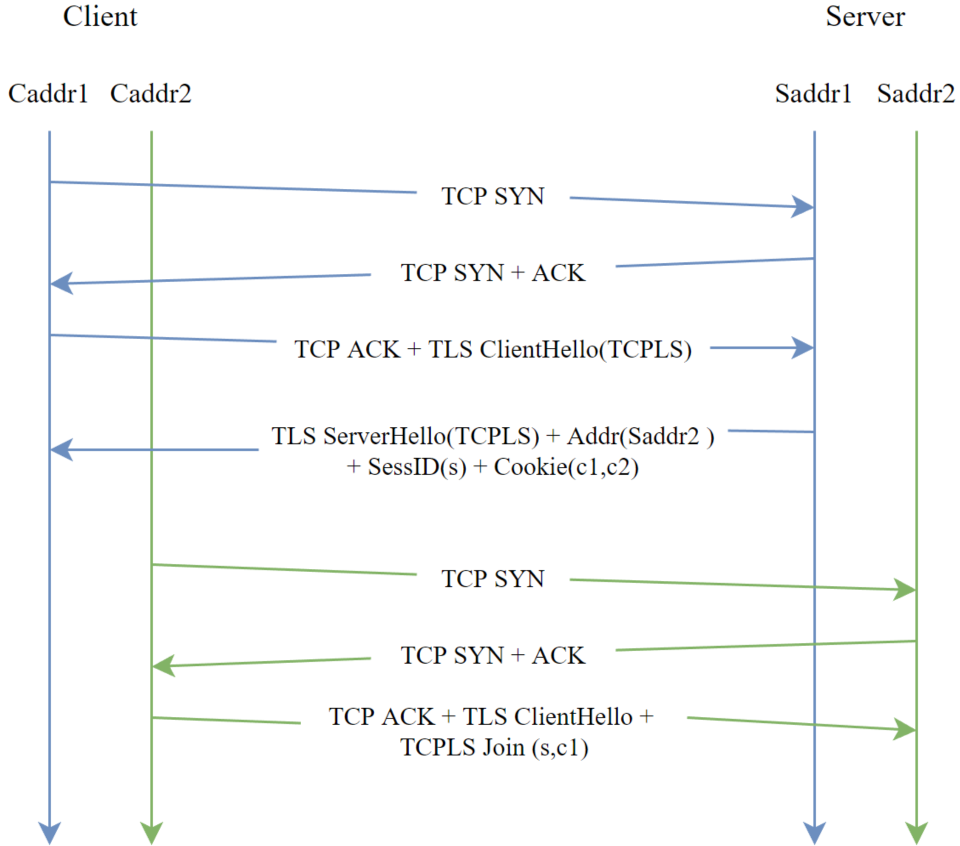


Figure 2.3: Joining a TCPLS Session

The aggregated mode can be used to schedule data over multiple streams mapped to several TCP connections. The records contain a sequence number so the receiver can reorder them and deliver the payload in order. This allows to combining the bandwidth of the different paths that are used to achieve higher throughput at the cost of losing the abstraction of using different streams for different purposes.

The application is responsible for the scheduling of the data over the different streams. This provides high flexibility in the way each underlying TCP connection is used but also adds the complexity of the scheduler to the application.

2.2 Evolution

Following the first version of TCPLS, the authors presented the protocol at the IETF [37]. This new version keeps the original idea of building a protocol using both TCP and TLS in order to provide advanced transport services to applications but uses TLS records in a different way. The session establishment remains essentially the same, leveraging TLS Extensions in the ClientHello and ServerHello records to signal their support of TCPLS.

This version does not rely on new types of TLS records to exchange TCPLS data. Instead, it uses a new basic unit of information: TCPLS frames. Frames are Type-Value messages placed inside TLS Application Data records and allow to exchange both application data and control information. The first byte of each frame indicates its type and the following bytes correspond to type-specific fields. A single frame can not span over multiple TLS records but one record may contain more than one frame. An example of a TLS record containing two TCPLS frames is presented in figure 2.4.

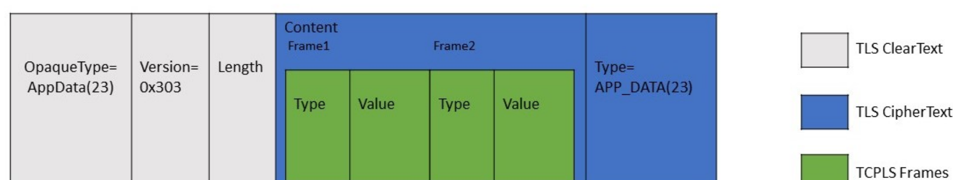


Figure 2.4: TCPLSv2 Frames format

2.2.1 Updated Transport Features

The transport features provided by the TCPLSv1 had to be updated to make use of frames instead of records. We present in this section how each of the services proposed by TCPLS has been changed.

Stream Multiplexing

Application data is exchanged through bidirectional streams using Stream frames. Each stream is identified by a 32-bit value that is explicitly sent in each Stream frame. The last bit of the identifier indicates whether the stream has been opened by the client (0) or the server (1). A stream frame can have two different types, 0x02 or 0x03. The latter is used to indicate that the stream ends with this

frame. As in the first version, streams opened on the same TCP connection can suffer head of line blocking but they can be spread among multiple TCP connections.

Unlike the streams of the first version, these streams are not tied to one connection and streams frame can be sent over any TCP connection attached to the TCPLS session as they contain the identifier of the stream.

Connection Multiplexing

Joining new TCP connections to a TCPLS session is somewhat similar to the first version but it uses fewer TLS Extensions and replaces them with frames. A TCPLS endpoint can use New Address and Remove Address frames to indicate the IP addresses it is currently willing to use for a session. Servers can issue New Token frames containing a token and a connectionID. In order to join a session, a client includes a token previously received in a New Token frame with the TCPLS Join Extension when sending its TLS ClientHello on a TCP connection using an address advertised by the server. The server verifies the token and adds the corresponding connectionID to the session. As only the token which is only used once is included, it is not possible to correlate the connections.

As before, it is possible to migrate a connection thanks to ACK frames. These frames indicate the highest record sequence number received on a specific connection identified by its connectionID. It allows to receiver to know what frames have been successfully received even over a failed connection and hence determine the ones that were lost. To migrate a TCP connection, an endpoint only needs to send the frames on other connections using the acknowledgements to retransmit records that might have been lost if the connection was aborted.

Multipath

Using multiple TCP connections at the same time allows to benefit from different paths. The scheduler is still defined by the application and can be even more flexible by allowing stream frames belonging to the same stream to be sent over different TCP connections. The receiver uses the offset value present in the stream frames to deliver data in order to the application.

2.2.2 Record protection

As multiple frames from different streams can be contained in a single TLS record, the streamID can no longer be used to make the record nonce unique. Instead of using one cryptographic context per stream, the new version uses one context per

TCP connection. The construction of the nonce for a TCPLS record is represented in figure 2.5. It uses the AEAD IV of the first TLS handshake and xors the first 32bits with the connectionID and the last 64bits with a record sequence number kept implicit and maintained by both the client and the server.

Figure 2.5: AEAD Nonce of TCPLSv2 Streams Records

2.2.3 Comparison

Table 2.1 summarizes the differences between the two versions. The key points are that using TCPLS frames that are inside TLS records instead of relying on TLS records with new types allows more flexibility by sending multiple frames inside the same record while also requiring fewer modifications to TLS. In addition, making cryptographic contexts connection-based instead of stream-based makes finding the right context easier for the receiver as it just has to look at the connection on which it receives the record instead of relying on trial and error. Furthermore, not having the streams tied to the cryptographic context enables the stream to be sent over different connections at the cost of explicitly indicating the stream identifier. In addition, it means that lost records have to be re-encrypted using the context of the connection increasing latency but making sure that connections are not correlated.

	TCPLSv1	TCPLSv2
Basic Unit	TLS Record	TCPLS Frame
AEAD Nonce Derivation	One context per stream	One context per connection
Connection Multiplexing	✓	✓
Join Mechanism	Relies on TLS Extensions	Relies on TCPLS Frames
Stream Multiplexing	✓	✓
Multipath	✓	✓

Table 2.1: Comparison between TCPLSv1 and TCPLSv2

Chapter 3

TCPLS evaluation

This chapter discusses the evaluation of the TCPLS prototype [35] with some experiments that were not undertaken in previous papers such as stream multiplexing and portability to mobile devices. We based our evaluation on a forked version of the protocol updated up to the commit of the 14th of December 2021 [1], the latest at the moment of writing, and including miscellaneous fixes regarding, among other, streams buffers.

3.1 Stream multiplexing

The first thing that we consider is the multiplexing of streams and the impact on the throughput that can be achieved. In the current TCPLS prototype, each stream uses a different AEAD IV derived from the TLS one. This means that upon reception of a message, the receiver has to find which context has to be used to decipher the data which is not simple as the stream ID is itself encrypted.

3.1.1 Decryption cost

The cost of using the wrong context to decipher a record should be similar to the time taken to successfully decipher a record in order to avoid eventual attackers to use timing attacks [41] in which they can know whether the decryption has succeed based on timing measurements. Hence, we started by observing the behaviour of the protocol when the decryption algorithm is applied multiple times before each message is delivered. This provided a base to evaluate the multiplexing mechanism.

[1] https://github.com/pluginized-protocols/picotcpls/tree/58c25481e40b979500685fc317e36aae7fcd4793

3.1.2 Impact of the scheduler

The scheduler used to send the data over each stream can impact performance. The pseudo-code of the reception handler used in the first prototype is described in the algorithm 1. It consists of two different functions *data_process* which is called by the receive scheduler and *try_decrypt_with_multistreams* which is called by *data_process* as long as it can successfully decipher data. The function tries to decipher with all opened streams for this connection in order. Switching the AEAD context between each stream but keeping the same context as long as it successfully deciphers records. Upon failure, it tries the next stream and so on. Finally, it uses the initial context and tries to decipher control messages that do not belong to any stream like stream attach messages. From this behaviour, we can compute how many attempts will be needed for a specific scheduler.

Best case scheduler : Sequential

The scheduler expected to have the highest throughput is a sequential one, each stream has a priority equal to the order in which it was opened. This way, the decryption only fails when there is no data to send on the first stream. There should not be any notable impact on performance as long as there is data to send on the highest priority stream.

Worst case scheduler : Round-Robin

The worst-case happens when records are received from streams in order starting from the last stream created to the first corresponding to a round-robin scheduler. This would cause the receiver to test all contexts before finding the right one and this for every record. The performance should decrease linearly with the number of streams and using N streams would be the same as doing the decryption N times.

However, when the streams are ordered from the first created stream to the last, the receiver would be able to alternate between a successful decryption and failed one. In this case, the number of decryptions should be 2 for most records as soon as the number of streams becomes greater than 2.

3.1.3 Methodology

The test case used is a file transfer between two hosts directly connected via an Ethernet cable. The CPU used on both hosts is an Intel(R) Xeon(R) Silver 4314 CPU 2.40GHz and they each have 32GB of RAM. They run Ubuntu 20.04.4 with a Linux 5.13.0-23-generic kernel. Both are equipped with Mellanox Technologies MT27800 Family [ConnectX-5] network interface cards. We used the prototype

Algorithm 1 decryption with multistreams

```
function data_process(connection, input_size, input)
    input_off = 0
    while progress do
        try_decrypt_with_multistreams(input, &input_off, input_size)
    end while
    return TCPLS_OK
end function

function try_decrypt_with_multistreams(input, input_off, input_size)
    for Stream S ∈ streamslist do
        if S == rcv_con then
            Remember_aead = decrypt.aead
            decrypt.aead = S.aead
            do
                consumed = input_size − *input_off
                received = ptls_receive(input + *input_off, &consumed)
                *input_off+ = consumed
            while received && *input_off < input_size
            decrypt.aead = Remember_aead
        end if
    end for
    if BAD_RECORD_MAC then
        do
            consumed = input_size − *input_off
            received = ptls_receive(input + *input_off, &consumed)
            *input_off+ = consumed
        while received && *input_off < input_size
    end if
end function
```

of the first version of TCPLS using one host as a client and the other one as a server.

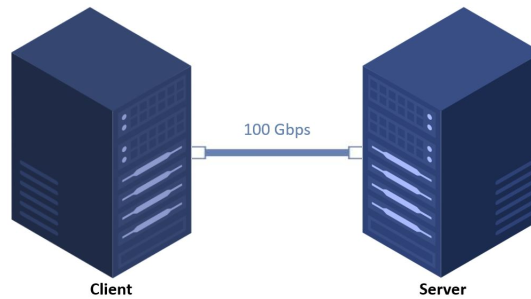


Figure 3.1: Stream multiplexing measurements setup

We measured the time to send 1GB of data from RAM by the server and directly consumed by the client from the moment the handshake is done until the client received the last stream close message. We chose to present the result as the ratio between the time taken compared to the base case in order to see the impact more clearly. We defined the base case as the average time needed to complete the transfer when using a single stream with no additional decryption. The cipher algorithm used was AES128 GCM SHA256. We tested both stream modes : aggregated and per stream buffer but as there was only one TCP connection, no reordering was needed and there was no significant difference in the measures so we only present the result with stream-based buffers.

3.1.4 Results

The results of performing multiple times the decryption operation are shown in figure 3.2. Each scenario has been run five times. We observe that the increase in time is proportional to the number of times records are deciphered and each additional decryption degrades performance by about a quarter compared to using only one decryption. This indicates that the decryption of records takes about 25% of the total time in the transfer.

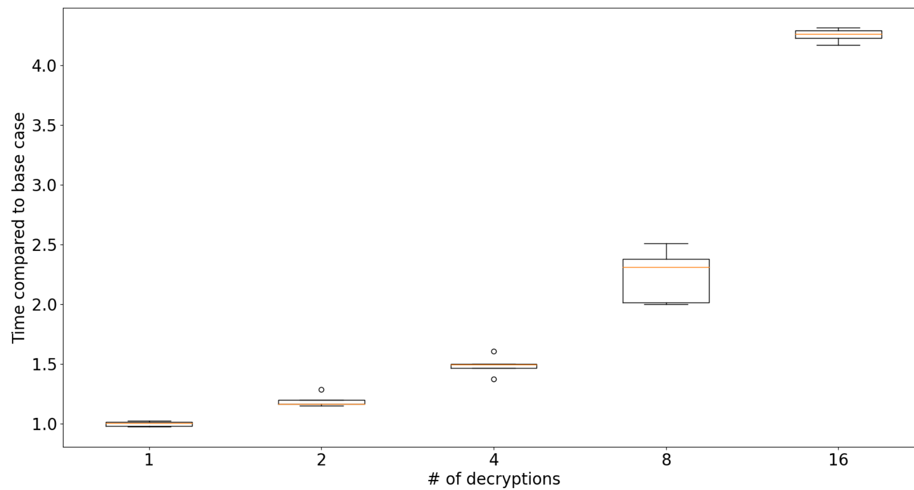


Figure 3.2: Performance using multiple decryption

The performance is very similar to using the round-robin scheduler starting from the last stream created to the first stream created (figure 3.3) but the overhead of creating, managing and closing the streams seems to make the transfer even slower.

When the scheduler starts from the first stream created to the last (figure 3.3), the performance decreases from 1 stream to 2 as in the reverse order but afterwards, they tend to increase a bit. This could be because when there is a lot of data in the receive buffer, there will be as many records decrypted when calling the function trying all streams as there are streams. This means that decryptions are made in larger bulk and could benefit from better cache locality.

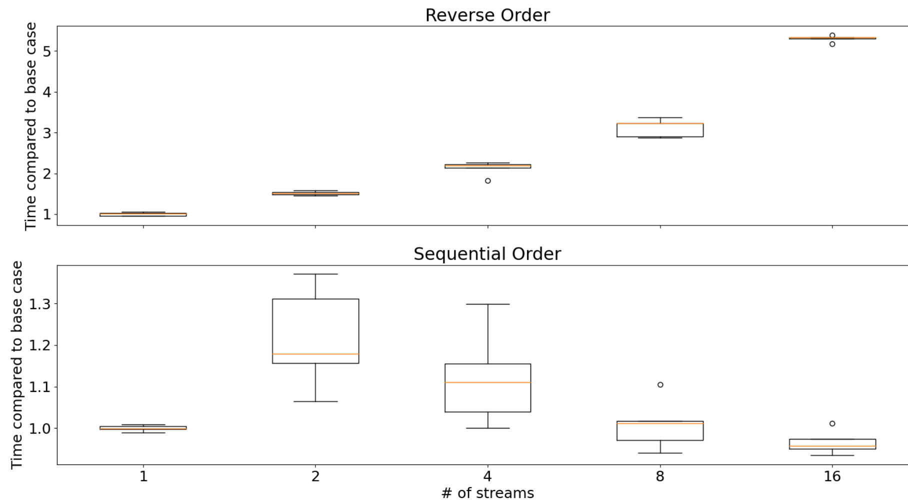


Figure 3.3: Performances with round-robin schedulers

3.2 Android support

As many of the services proposed by TCPLS like handover and multi-homing are particularly useful for mobile devices, we discuss in this section the portability of the protocol to such devices. We built an Android picotcpls library leveraging the Android Native Development Kit (NDK) [38] and developed an application using this library to exchange TCPLS traffic with a classic TCPLS server. This application has two goals:

- Show how the TCPLS prototype can be adapted to Android devices
- Prove that TCPLS can be used over real-world mobile networks despite middleboxes

3.2.1 Picotcpls Android library

In order to build the application, we first made the C code from the prototype runnable on Android devices. The NDK is a set of tools that can be used to compile C code into Java native libraries that can be bundled into an Android Package Kit (APK). These libraries can be used by an application through the Java Native Interface (JNI) [39]. The JNI provides a programming interface that allows defining and importing Java Objects in C and to export C functions that

can be loaded and executed by a Java Virtual Machine (JVM).

The first step to build our library was to adapt the existing code to the NDK. Indeed, the kit uses a special toolchain to compile the code that uses its own standard libraries that do not support all function calls. We only had to modify about 10 lines of code to replace unsupported calls such as bzero(3) [40] which is a legacy function that is used within picotcpls.

Next, we developed an additional library leveraging the JNI to enable a Java application to have access to TCPLS. Instead of providing access to the complete TCPLS API that would require the adaptation of the structures used such as the tcpls context, we decided to only allow the application to run specific integration tests like doing a TCPLS handshake with a server or transferring a file using multipath. Our library is based on the existing CLI and provides the same set of tests in about 900 lines of code.

The last step was to package all the libraries together so they could be used by our application. This includes the TCPLS JNI library, the picotcpls core libraries as well as all the dependencies like OpenSSL that also need to be compiled with the NDK.

3.2.2 TCPLS Mobile Application

We built an Android application including a JNI module making use of the picotcpls Android library to allow users to exchange TCPLS traffic. The different components of the application and how they interact are represented in figure 3.4.

The application consists of several Android activities allowing to choose between the integration tests, testing the classical handshake, the 0-RTT handshake, file transfer, etc. It is also possible to change the IP addresses and the port used to join the server as well as toggle multipath. The Java code represents around 300 lines of code for the activities and additional classes to represent the state and trigger messages. This version is compatible with ARM64-v8a devices using a minimal Android API level of 28 corresponding to Android 9 and above.

3.2.3 Validation

We validated the application by running the tests in different environments. We used a Nokia 3.4 with 3GB of RAM to run the application and run the server on a virtual machine hosted by OVH in their Canadian data center. The server runs CentOS Stream 8 with the Linux 4.18.0-348.el8.x86_64 kernel. We also used a

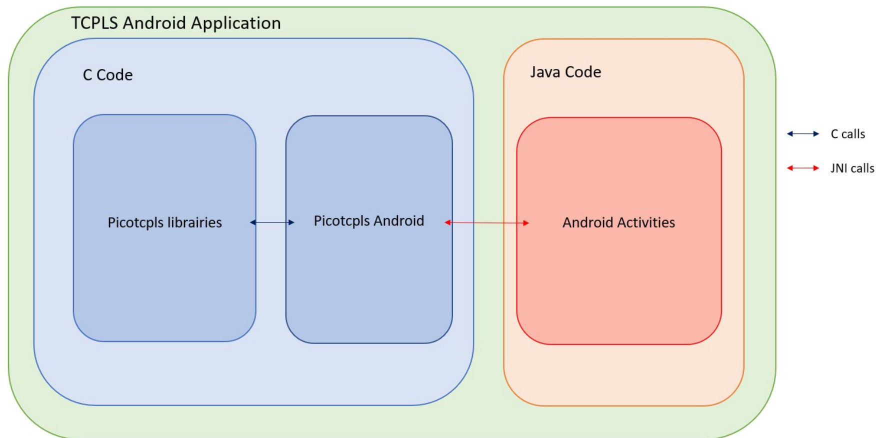


Figure 3.4: TCPLS Application structure

Voo Technicolor model TC7210.V as WiFi router. We had three paths to reach the server: WiFi with both IPv4 and IPv6 and LTE represented in figure 3.5 and used them in three different scenarios :

- Single path transfer over Wi-Fi (IPv4 only)
- Multipath aggregated transfer over Wi-Fi (IPv6 & IPv4)
- Single path transfer over LTE

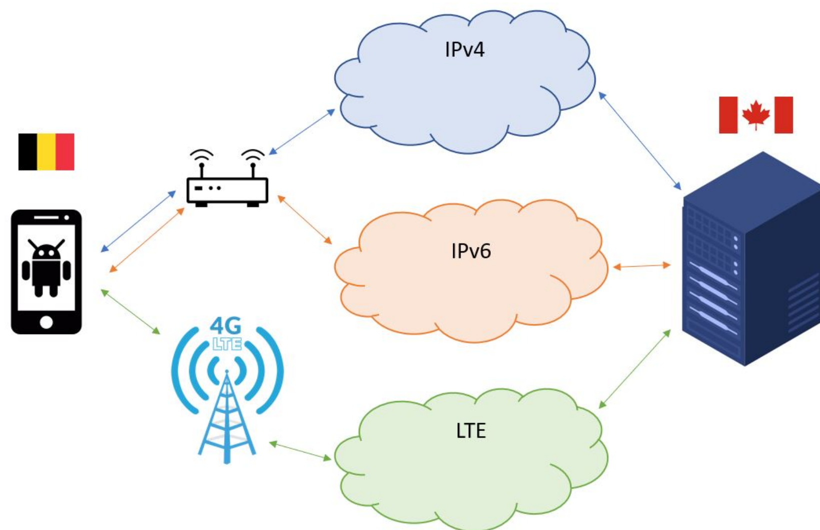


Figure 3.5: TCPL Application testing environments

For each case, we measured the time taken to transfer a 10 MB file stored on the server to the phone. The results for performing 10 times the transfer are shown in figure 3.6. In our configuration, using the aggregated bandwidth over IPv4 and IPv6 can slightly improve performance but also increase the variations. The LTE seems to yield better results but it probably depends on the WiFi and LTE signal quality. Beyond the performance, this experiment proves that the protocol can be used within large networks including mobile ones which are prone to contain lots of middleboxes [31].

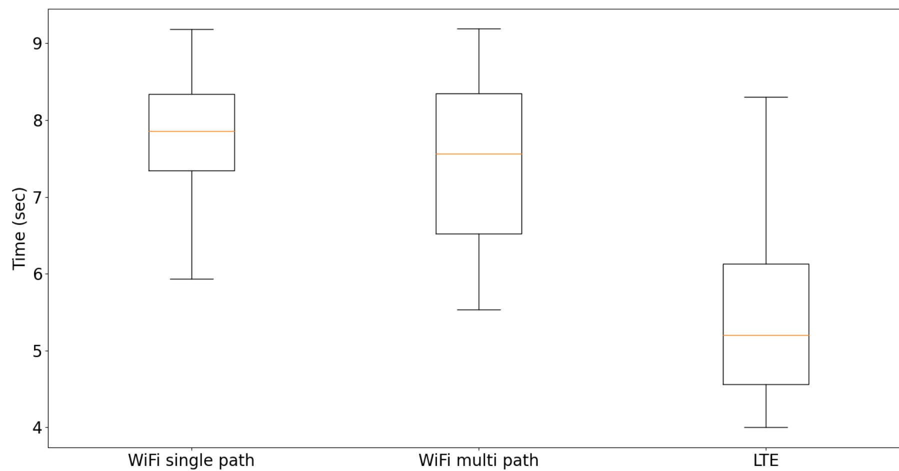


Figure 3.6: Time to transfer 10 MB

Chapter 4

TCPLS extensions

In this chapter, we propose extensions to the TCPLS protocol to exchange information about the quality of TCP connections like the RTT, the number of losses and retransmissions as well as to detect NAT on a path and to provide flow control. These are all based on the first version of TCPLS and have been implemented on our fork of the initial prototype but all the control records we designed could be included in the IETF version as frames with only minor modifications.

4.1 TCP information exchange

As the application is responsible for the scheduling of the streams over the TCP connections, it should be able to probe the connection information to make the best choices possible depending on its purpose and the state of connections. Because the underlying sockets are directly exposed to the application it is already possible to use the TCP Info socket option to gather information about one side of the connection. In this section, we propose two extensions to TCPLS that allow applications to estimate the RTT of a connection and get the TCP info from a peer.

To be able to exchange this information, we propose two pairs of TCPLS Control records, Ping Info / Info Reply and Ping RTT / RTT Reply. The Ping Info record only contains a connectionID requesting the corresponding tcp_info. Upon reception of such a message, an Info Reply is sent back with the same connectionID and the tcp_info structure. Both records trigger a callback function so the information can be delivered to the application, analyzed and used to adjust the scheduler or get rid of connections with poor quality. Applications that are not interested in the whole tcp_info structure can use the Ping RTT which contains a 16 bytes timestamp after the connectionID. This timestamp is echoed back in a RTT Reply

record and is received by the application through a callback function which allows it to estimate the RTT by comparing the timestamp value with the current time. The record format of all these records is represented in figure 4.1

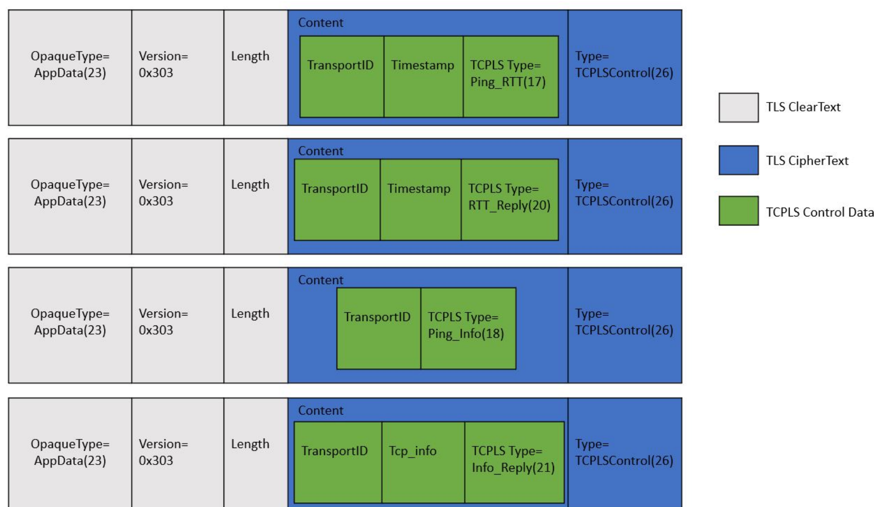


Figure 4.1: Ping RTT / RTT Reply and Ping Info / Info Reply record format

The support of these new records and the new API calls used to trigger the ping messages fit in around 150 lines of code. We tested the probing of TCP connections using IPMininet [49] to simulate an environment with multiple paths between the client and the server offering different quality of connections. The network is represented in figure 4.2. The IPv4 path suffers from an additional delay while the IPv6 one presents losses. We used Ping RTT messages in order to estimate 10 times the RTT on both paths and obtained the results shown in figure 4.3. We can see that the RTT when using the IPv4 path is over 0.4 sec corresponding to two times the delay which is logical as both directions are affected by the delay. The IPv6 does not have an additional delay and the RTT measured is way lower.

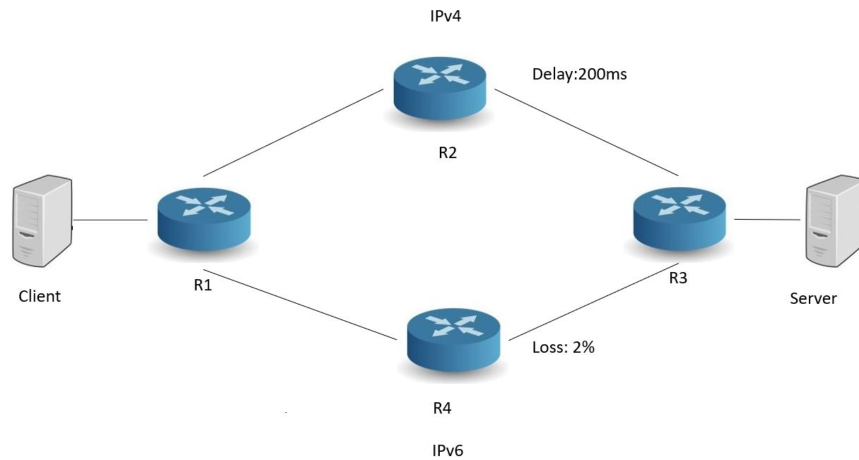


Figure 4.2: Multipath IPMininet Network

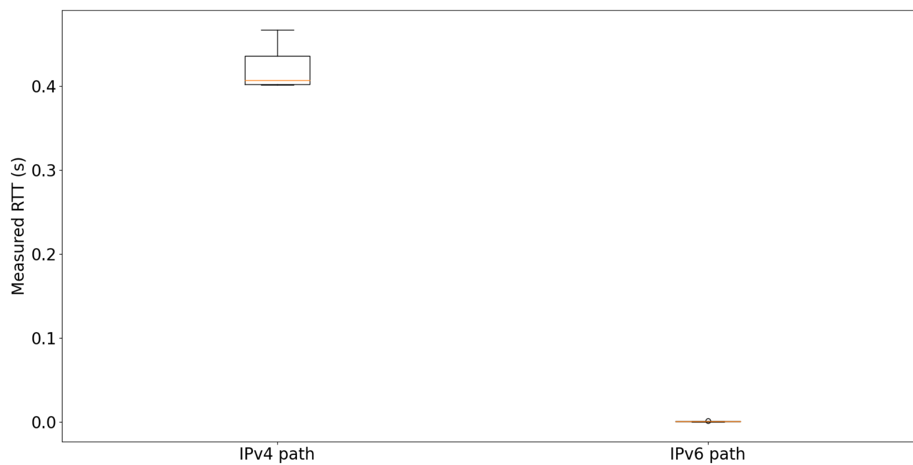


Figure 4.3: RTT estimation using Ping RTT

4.2 NAT detection

Middleboxes that break the end-to-end principle of TCP such as NATs can prevent the TCP options exchanged over TCPLS to be effectively used. This section explains how we extended TCPLS to be able to detect NATs, the drawbacks and the advantages of the TCPLS approach compared to other existing techniques and how we validated it.

4.2.1 Principles

TCPLS does not provide any control on IP TTL of ICMP packets and hence cannot be used to detect interference with TCP options or sequence numbers. However, it has access to the IP address and ports used by the underlying TCP session which is sufficient to be able to detect NATs.

The idea is to send a special TCPLS control message which, when received triggers a response quoting the address and port used by the connection on which it was received. The sender can use this message to detect anomalies between the address and port it uses and the ones that are perceived by the receiver. Figure 4.4 illustrates the mechanism when there is a change in the TCP port.

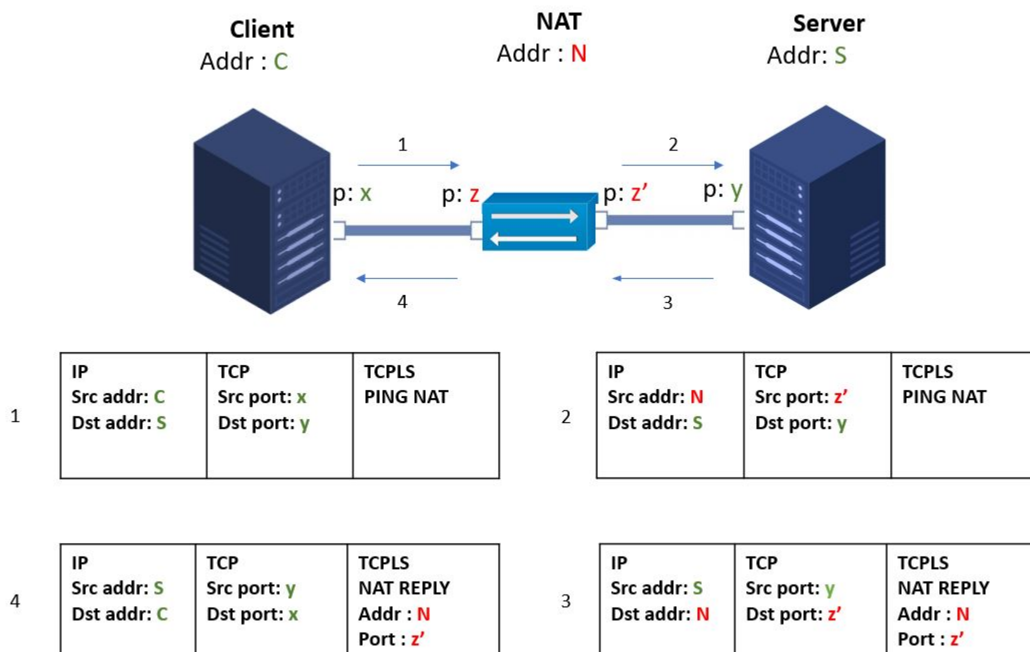


Figure 4.4: TCPLS NAT detection

4.2.2 Implementation

We added around 150 lines of code to the TCPLS prototype to support two new control records:

- Ping NAT: Record requesting the receiver to send the port and address used

for a specified TCP connection. Upon reception of such a message, a NAT REPLY is sent back and a call is triggered.

- NAT Reply: This record contains the requested information about the connection specified.

We also extended the TCPLS API with a function *tcpls_ping_nat(tcpls, transportid)* which, provided a context and a transport ID, sends the corresponding Ping NAT message. When a NAT Reply is received, it triggers a callback to transfer the information about the connection to the application which can compare it with its own and decide how to react accordingly.

4.2.3 Packets format

As for other TCPLS control records, the Ping NAT and NAT reply records are inside TLS application data with the encrypted type being TCPLS Control and the TCPLS message type being 19 for Ping NAT and 22 for NAT reply. The Ping NAT message contains only the transport ID of the connection to get the information from. The NAT reply message contains the transport ID of the connection as well as the address family, the port used by the connection and the IP address. All fields are encoded in network-byte order.

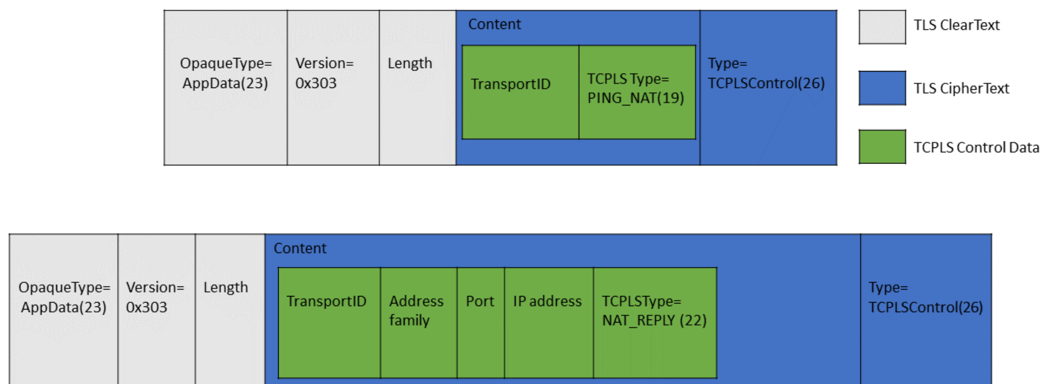


Figure 4.5: Format of Ping NAT and NAT reply messages

4.2.4 ICMP based middlebox detection

The description of ICMP, in RFC792[42], states that ICMP time-exceeded messages should contain the IP header and the first 64 bits of packets that have a time to live (TTL) equal to 1. Later, RFC1812 [43] recommended quoting the entire IP packet, this is not supported by all routers but it is becoming more and more deployed.

Based on these quoted packets, Detal et al. [44] developed tracebox, a traceroute extension able to detect middleboxes by observing changes between the TCP headers sent and the quoted ones. Figure 4.6 illustrates the detection of a middlebox using tracebox.

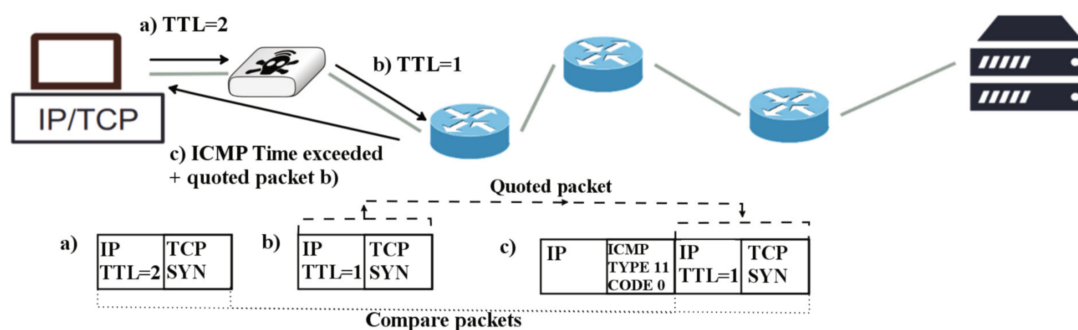


Figure 4.6: Tracebox with middlebox (from [45])

4.2.5 Comparison with ICMP based detection

One of the advantages of using TCPLS instead of ICMP is that many NATs also translate the headers of ICMP packets. Tracebox avoids this by looking for Application-level Gateways (ALGs) [22] that are often combined with NATs to support protocols with IP address dependencies such as FTP. This solution is not perfect as it requires a router compliant with RFC1812 on the path but when the tracebox experiment was made, this represented 80% of the paths.

TCPLS requires both endpoints to cooperate in order to be able to detect modifications which is not the case of tracebox that relies on routers on the path. By working at the IP layer, tracebox is able to detect much more anomalies as it had access directly to the TCP header.

4.2.6 Validation

We validated the detection using IPmininet in order to simulate a network with different routers in which we introduced a simple NAT configured to mask the IP of the traffic coming from R1 with its own address and forward it to R2. The network is represented in figure 4.7.

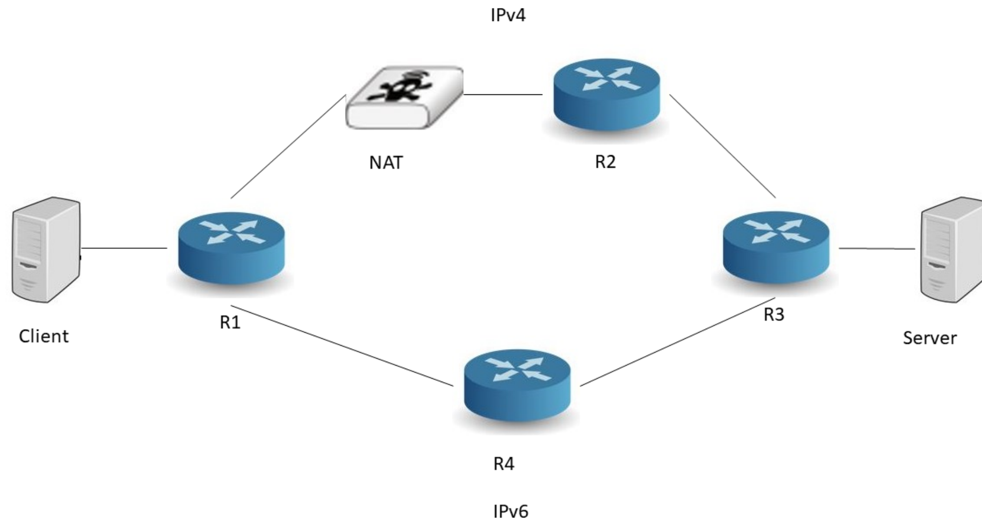


Figure 4.7: IPMininet network with a NAT

The client has two different paths to reach the server, one using IPv4 and the other using IPv6. The NAT is placed on the IPv4 path and is configured to change the address of TCP packets sent by R1 that it forwards. The client can open a connection with the server on both paths and send Ping NAT messages. On the IPv6 path, the underlying TCP connection is not modified between the client and the server but on the IPv4 side, the server only sees the address of the NAT. When the client receives the NAT reply from the server on each path, it can detect that the IPv4 address does not correspond to the one bound to his socket while the IPv6 is the same.

4.3 Flow shaping

Sending as much data as possible might not be the best way to make use of a connection or a network. HTTP/2 uses flow control at both the connection and

stream level to ensure that the sender does not overwhelm the receiver and to prevent streams of a single connection from interfering with each other. This leverages a flow control window indicating the number of bytes that can be transmitted. This window can be updated with specific frames. Another example is traffic shaping and traffic policing which are used to provide differentiated services [50]. We added the possibility for applications to limit the TCPLS traffic by advertising a maximum rate at which a sender can transmit data. For example, this allows a mobile application to limit the usage of cellular networks to avoid extra fees.

4.3.1 Principles

We decided to introduce flow shaping only at the connection level and not at the stream level as the application can schedule the streams over the different connections to define its own control mechanism with full knowledge of the requirements of each stream and the capability of the connections. The receiver can inform the sender of a limit on the number of bytes that can be sent over a connection by sending a Flow Limit record associating the connectionID with a maximum number of bytes per second that can be sent. The exact format of the record is shown in figure 4.8. After receiving such a message, the sender should not send stream data at a higher rate than the one indicated by the receiver. Regardless, it is still possible to update the rate by sending additional Flow Control records. If the application attempts to send data when the limit is reached, the send call fails with a special value indicating that the data cannot be sent. Control messages are not impacted by this limit.

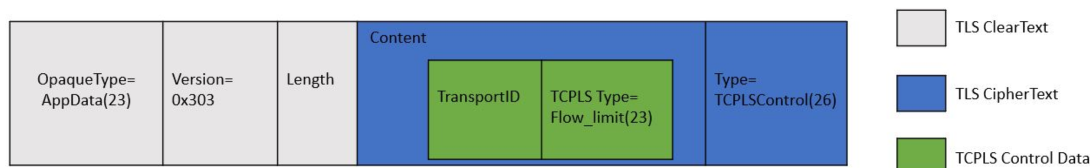


Figure 4.8: TCPLS Flow record format

4.3.2 Implementation

We implemented the parsing and sending of the Flow Limit as well as the actual limitations to sending in less than 100 lines of code. In order to respect the limit, we used the token bucket principle [51]. If the receiver signals a limit for a connection, it becomes marked as limited by the sender with the associated maximum rate. Limited connections have a pool of tokens they can use to send

bytes. At each transmission, the number of tokens is refilled by adding an amount corresponding to the maximum rate multiplied by the time since the last refill up to a maximum. Afterward, the number of tokens is reduced by the number of bytes to send provided that it is enough. If it is not the case, the send call fails and returns an error code indicating that the limit has been reached. The pseudo-code of this mechanism is presented on algorithm 2. We used an arbitrary limit of four records for the numbers of tokens a single connection can hold at any time. We leave the negotiation of the token bucket parameters for further extensions.

Algorithm 2 Flow Control with Token Bucket

```
function tcpls_send(stream, input, input_size)
    connection = connection_get(stream)
    if connection− > limited then
        refill_tokens(connection)
        if connection− > tokens < input_size then
            return Limit_Reached
        end if
        connection− > tokens− = input_size
    end if
    return ptls_send(stream, input, input_size)
end function

function refill_tokens(connection)
    time_diff = now − connection− > last_refill
    connection− > tokens+ = time_diff ∗ connection− > token_rate
    if connection− > tokens > Max_Tokens then
        connection− > tokens = Max_Tokens
    end ifconnection− > last_refill = now
end function
```

4.3.3 Validation

We tested this feature with the same hosts and in the same environment as the one used for stream multiplexing in section 3.1.3. We measured the evolution of the goodput over time during a 1GB transfer in which we introduce a limitation after 0.3 sec. The server reads bytes from a file stored in RAM and the server writes the number of bytes received on the file system whenever there is new data. The results and shown in figure 4.9. We used different values for the limit of each flow. All the limits are represented in a dashed line with the same color as the flow except for flow 3 which does not have a limit. The flows 0 and 1 set a limit

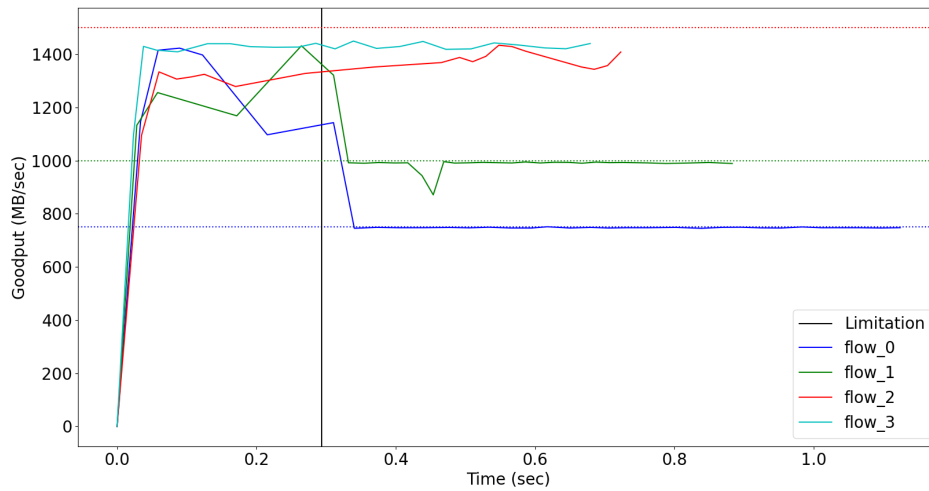


Figure 4.9: Evolution of goodput with Flow limit

of respectively 750 and 1000 MB/sec. These values are lower than the maximum goodput that can be achieved and so it drops to the value of the limit. Flow 2 uses a limit of 1500 MB/sec which is higher than the maximum possible goodput, it is not really impacted by the limitation and it remains close to the limitless flow 3.

Chapter 5

TCPLS use case: Tunneling Internet protocols inside TCPLS

In this chapter, we present how we built and evaluated tunneling services over TCPLS. Tunnels are particularly useful for devices that can be attached to untrusted networks such as phones or laptops. Current solutions like TLS [17], DTLS [23] or IPSec [52] provide encryption and authentication to protect those devices from eavesdropping and message modification. However, these protocols do not provide support for multi-homing or connection migration which could be beneficial, especially for mobile devices. Piraux et al. [53] proposed a design of tunnel for UDP and TCP over Multipath QUIC [54]. Our goal was to adapt the stream mode to tunnel TCP traffic from a client to a remote server using an intermediate concentrator as shown in figure 5.1.

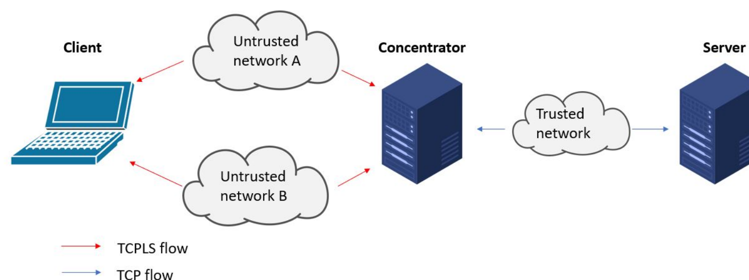


Figure 5.1: Example Environment

5.1 Opening a TCPLS Tunnel

The tunneling protocol uses TCPLS streams to carry the data exchanged over the TCP connection by sending specific messages over a stream. The messages contain the IP address and port of the final destination and the concentrator is responsible for opening the corresponding TCP connection. Once the connection is opened, the client is notified and all the data sent on the stream is transferred to the TCP connection and the other way around. The exact format of the messages is described in section 5.2.

5.2 Messages format

Messages used to initiate a tunnel are sent over a TCPLS stream in the form of *Type* , *Length*, *Value* (TLV) tuples (figure 5.2). The *Type* is encoded on one-byte and indicates the type of the message. The *Length* field contains a one byte value indicating the length of the *Value* field. The *Value* field contains different values depending on the type of the message. A TLV can have a length equal to zero in which case it does not contain any value and only the type provides useful information.

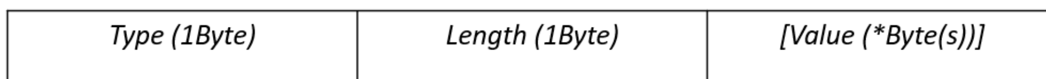


Figure 5.2: TCPLS Tunnel Messages Format

The protocol uses the following types of messages :

- TCP Connect (0x00) : Used to establish a TCP connection on a TCPLS stream.
- TCP Connect OK (0x01) : Indicates successful establishment of a TCP connection.
- ERROR (0x02) : Indicates an error during the connection establishment.
- END (0xff) : Marks the beginning of the bytestream. It is used to indicate that it is the last TLV sent on this stream.

5.2.1 TCP Connect

TCP Connect messages are used to indicate the destination of the tunnel. The format is represented in figure 5.3. They contain the port and the IP address that need to be used in the *Value* field. The IP address needs to be encoded as an IPv6 address. IPv4 can still be used but they have to be encoded in the IPv4-Mapped IPv6 format described in RFC4291 [55]. This makes the *Length* field always equal to 18 (2 bytes for the port number and 16 for the IP address).

TCPLS Connect (0)	Length (18)	Port Number
IP Address		

Figure 5.3: TCP Connect message format

5.2.2 TCP Connect OK

TCP Connect OK is used to confirm that the TCP connection has been successfully established. It does not contain a value meaning that its *Length* should always be 0.

5.2.3 Error

Error messages are used to signal errors that can occur while establishing the tunnel. The *Value* of an Error TLV is a 2 bytes code indicating the kind of error. We use the following types of error :

- Protocol Violation (0x0000): Generic error code used when a non-conforming behaviour not corresponding to another error is encountered.
- Malformed TLV (0x0001): Used upon reception of an invalid TLV.
- Connection Failure (0x0002): Used if the TCP connection fails to be established.

As there can be multiple error messages, the sender should send an End TLS after the last one and close the stream.

Error (2)	Length	Error Code

Figure 5.4: Error message format

5.2.4 End

The End message indicates the end of the transmission of TLVs. All subsequent bytes are part of the tunneled byte-stream. It does not contain any value and has a 0 *Length*.

5.3 Prototype

We implemented a concentrator supporting the protocol and capable of opening TCP connections on the behalf of clients. We also implemented a client application leveraging the tunnel to exchange data with a TCP server. The client and the concentrator represent around 1.5k lines of C code.

In order to handle different clients, the concentrator maintains a linked list with structures containing information for each opened TCPLS session. These structures are composed of the state of the connection, the TCPLS context, the buffers associated to the connection, the socket of the TCP connection used for TCPLS traffic, the one used to reach the final destination, the streamID, the connectionID, the amount of data sent in both directions and the time of the beginning of the connection. The concentrator listens on different sets of sockets, first, it waits for new connections on one of its IP address, then it exchanges TLVs on the opened TCPLS sessions that does not have an opened tunnel to create new TCP connections and, when the tunnel is open, it needs to transfer the data received on either side to the connection corresponding to the other.

In addition, we provide a simple TCP client and server that were used to perform tests and several scripts that can be used to reproduce our experiments. Everything is available in a dedicated folder inside our picotcpls fork.

5.4 Evaluation

In this section, we present the experimental environment and the use cases we used to validate and evaluate our prototype.

5.4.1 Experimental environment

We performed the experiments with the same hosts as for the multiplexing evaluation described in section 3.1.3. Because tunneling protocol requires a client, a server and a concentrator, one of the hosts had to run the client and the server at the same time using different ports. The second host acted as a concentrator and redirected the traffic between the client and the server process. The environment is represented in figure 5.5.

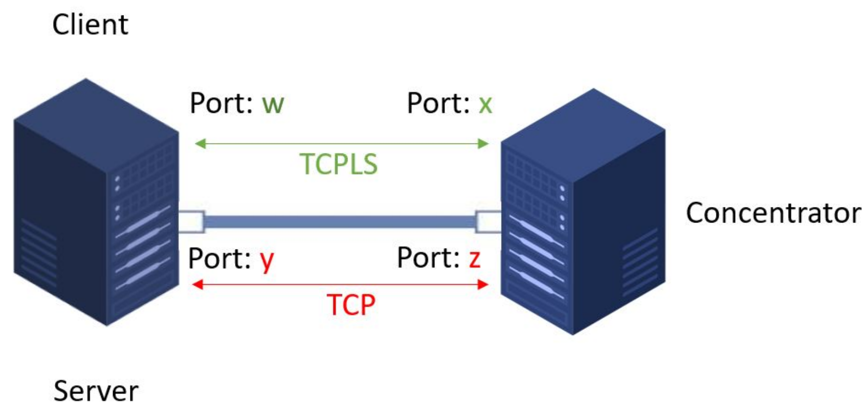


Figure 5.5: Experimental Tunneling Environment

5.4.2 Goodput

We measured the goodput that can be achieved by sending 10 Gbytes from the client to the server. The data is read directly from the stack of the client and the server simply discards it once it is received. We compared the differences between using a direct TCP connection, a direct TCPLS connection and a tunneled TCP connection. We repeated the transfer 100 times for each configuration. The results are shown in figure 5.6.

We can observe that the TCP connection have the highest goodput. As TCPLS provides encryption and authentication as well as additional headers, the performance drops from more than 20 Gbps to a little bit higher than 15. Adding the overhead of the tunnel further reduces the goodput because of the management of the different types of connections but it remains above 10 Gbps. This shows that using the tunnel does impact performance but not to a point where applications become unusable while enabling them to connect in a secure way to a distant server.

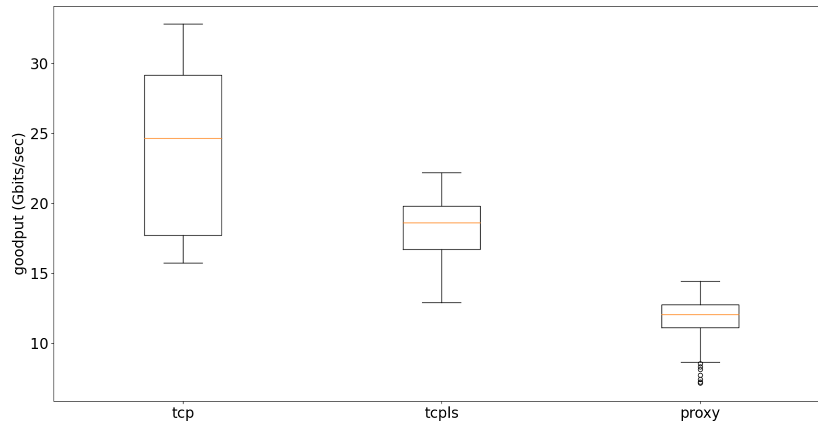


Figure 5.6: Goodput Comparison

5.4.3 Connection latency

Latency is one of the most important metrics for web search [56]. We measured the time required by a client to receive the first bytes of application data from a server. We compared the connection latency when using a TCP connection, a TCPLS connection and when using a TCPLS tunnel to open a TCP connection. The result for 100 connection openings are represented in figure 5.7.

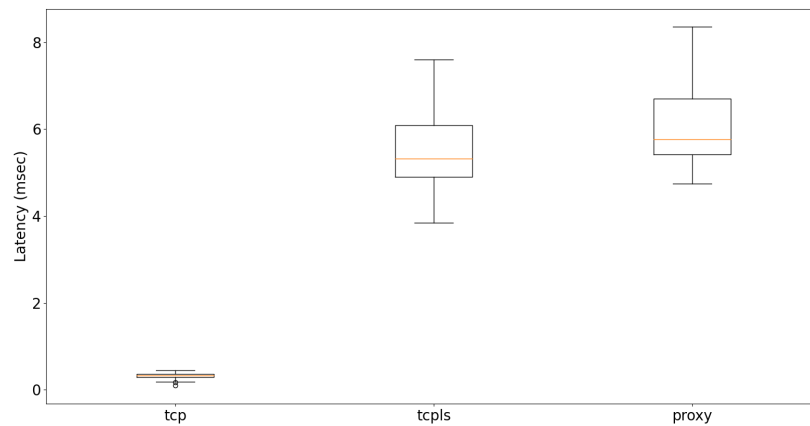


Figure 5.7: Connection Latency Comparison

Using only TCP yields the lowest results. The TCPLS handshake including the cryptographic context derivation has a huge impact on latency, going from less than 1 millisecond on average to more than 5. However, using the tunneled TCP connection is very close to the sum of using a TCPLS connection and a TCP one, even if it also has to send messages in order to open the tunnel.

5.4.4 Requests per second

The last metric we evaluated is the number of requests per second a server could serve. We simulated requests where the client sends 800 bytes of data corresponding to the typical size of an HTTP request header[57]. The server sends back a response upon reception of the request. We considered three sizes of responses: responses of 800 bytes matching the size of an HTTP response header, responses of 32KB corresponding to small objects and light web pages and finally, responses of 4MB representing larger objects. We measured the number of requests per second by using a client sending 1000 successive requests and waiting for the response before sending the next request. The results for each response size and the three protocols are presented in figure 5.8.

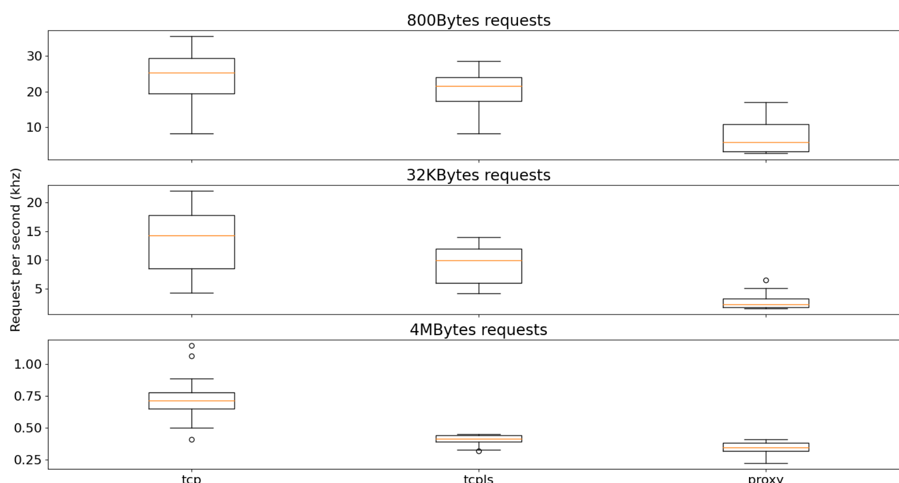


Figure 5.8: Request per seconds

TCP achieves the best number of requests per second but the discrepancy is less important than for latency or goodput. The tunneling overhead is quite important for small requests, halving the performance of using TCPLS but it is very small for bigger requests. Requests of 800 Bytes are treated two times faster than the 32 KBytes ones in all the different cases. We assume that this is because 32 KBytes

is too large to be sent in 1 burst and requires 2 RTTs. Requests of 4 MBytes are way bigger and require more RTTs making them slower.

Conclusion

In this thesis, we highlighted the importance of scheduling when using multiple TCPLS streams due to the decryption mechanisms by comparing the performance with different schedulers. We evaluated the portability of the TCPLS prototype by implementing a mobile application making use of the JNI to include our own picotcpls Android library built with the NDK.

We proposed extensions to the protocol to support the exchange of information regarding TCP connection, NAT detection and flow shaping. We implemented all these extensions and verified that they can achieve their goals through various test scenarios.

Finally, we designed and implemented a tunneling protocol leveraging TCPLS streams to enable hosts to establish secure byte streams regardless of their environment. We evaluated the performance of our prototype regarding the goodput, the connection latency and the number of requests per second that can be treated and showed that the overhead it introduces impacts the performance but not to a point where it is not usable.

Software artifacts

Our fork of the picotcpls prototype containing the tests for multiplexing, implementations and tests of the extensions is available at the following address: `https://github.com/AurelienBuchet/picotcpls`. It also contains the implementation of the tunneling protocol with the client and server inside the **/t/proxy** folder. The source code for the TCPLS Android application is available on the repository at `https://github.com/AurelienBuchet/-TCPLS-Android-`.

Future work

There are multiple aspects of this thesis that can be further explored as part of future work. First, we could specify the extension using the frames format of TCPLSv2 and try to include them as part of the protocol in the next versions. We could also design rules based on the information provided by the TCP info extension to provide adaptive connection and stream management. Finally, we could deploy an Android application at a larger scale to measure the performance of TCPLS within different environments with multiple users.

Bibliography

[1] Jacobson, V. (1988). Congestion avoidance and control. ACM SIGCOMM computer communication review, 18(4), 314-329.

[2] Paxson, V., Allman, M., Chu, J., & Sargent, M. (2011). Computing TCP's retransmission timer (No. rfc6298).

[3] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[4] Postel, J., "Internet Protocol", STD 5, RFC 791, USC/Information Sciences Institute, September 1981.

[5] Sanders, R. M., & Weaver, A. C. (1990). The Xpress transfer protocol (XTP)—a tutorial. ACM SIGCOMM Computer Communication Review, 20(5), 67-80.

[6] Clark, D. D., Jacobson, V., Romkey, J., & Salwen, H. (1989). An analysis of TCP processing overhead. IEEE Communications magazine, 27(6), 23-29.

[7] Thomson, M., & Turner, S. (2019). Using TLS to secure QUIC. Internet Engineering Task Force, Internet-Draft draft-ietf-quic-tls-24.

[8] Jacobson, V., Braden, R., & Borman, D. (1992). TCP extensions for high performance (pp. 1-28). RFC 1323, May.

[9] Kohler, E., Handley, M., & Floyd, S. (2006). Datagram congestion control protocol (DCCP) (No. rfc4340).

[10] Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., ... & Paxson, V. (2000). Stream control transmission protocol (No. rfc2960).

[11] Eddy, W. (2007). TCP SYN flooding attacks and common mitigations (pp. 2070-1721). RFC 4987, August.

[12] Ford, A., Raiciu, C., Handley, M., & Bonaventure, O. (2013). TCP extensions for multipath operation with multiple addresses.

[13] Raiciu, C., Paasch, C., Barre, S., Ford, A., Honda, M., Duchene, F., ... & Handley, M. (2012). How hard can it be? designing and implementing a deployable multipath TCP. In 9th USENIX symposium on networked systems design and implementation (NSDI 12) (pp. 399-412).

[14] Wischik, D., Raiciu, C., Greenhalgh, A., & Handley, M. (2011). Design, implementation and evaluation of congestion control for multipath TCP. In 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11).

[15] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M., & Tokuda, H. (2011, November). Is it still possible to extend TCP?. In Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference (pp. 181-194).

[16] Freier, A., Karlton, P., & Kocher, P. (2011). The secure sockets layer (SSL) protocol version 3.0 (Vol. 11). RFC 6101.

[17] Rescorla, E. (2018). The transport layer security (TLS) protocol version 1.3 (No. rfc8446).

[18] Dierks, T., & Rescorla, E. (2008). The transport layer security (TLS) protocol version 1.2. (No.rfc5246)

[19] Gutmann, P. (2014). Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). Request for Comments, 7366.

[20] Postel, J. (1980). RFC0768: User Datagram Protocol.

[21] Rescorla, E., & Modadugu, N. (2006). Datagram transport layer security. (No. rfc4347)

[22] Rescorla, E., & Modadugu, N. (2012). Datagram transport layer security version 1.2. (No. rfc6347)

[23] Rescorla, E., Tschofenig, H., & Modadugu, N. (2022). The Datagram Transport Layer Security (DTLS) Protocol Version 1.3. (No. rfc9147)

[24] Berners-Lee, T., Fielding, R., & Frystyk, H. (1996). Hypertext transfer protocol–HTTP/1.0.

[25] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (1999). Hypertext transfer protocol–HTTP/1.1.

[26] Belshe, M., Peon, R., & Thomson, M. (2015). Hypertext transfer protocol version 2 (HTTP/2).

[27] Stenberg, D. (2014). HTTP2 explained. ACM SIGCOMM Computer Communication Review, 44(3), 120-128.

[28] Peon, R., & Ruellan, H. (2015). HPACK: Header compression for HTTP/2. Internet Requests for Comments, RFC Editor, RFC, 7541.

[29] Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C., Zhang, D., ... & Shi, Z. (2017, August). The quic transport protocol: Design and internet-scale deployment. In Proceedings of the conference of the ACM special interest group on data communication (pp. 183-196).

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

[31] Wang, Z., Qian, Z., Xu, Q., Mao, Z., & Zhang, M. (2011). An untold story of middleboxes in cellular networks. ACM SIGCOMM Computer Communication Review, 41(4), 374-385.

[32] Carpenter, B., & Brim, S. (2002). Middleboxes: Taxonomy and issues (pp. 1-27). RFC 3234, February.

[33] Rochet, F., Assogba, E., Piraux, M., Edeline, K., Donnet, B., & Bonaventure, O. (2021, December). TCPLS: modern transport services with TCP and TLS. In Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies (pp. 45-59).

[34] Rochet, F., Assogba, E., & Bonaventure, O. (2020, November). TCPLS: Closely Integrating TCP and TLS. In Proceedings of the 19th ACM Workshop on Hot Topics in Networks (pp. 45-52).

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

[36] Eggert, L., & Gont, F. (2009). Tcp user timeout option. RFC 5482, March.

[37] Piraux, M., Bonaventure, O., Rochet, F. (2022, March). TCPLS: Modern Transport Services with TCP and TLS. Internet-Draft draft-piraux-tcpls-01. Internet Engineering Task Force.

[38] Android NDK. https://developer.android.com/ndk.

[39] Java Native Interface Specification. https://docs.oracle.com/javase/7/docs/technotes/guides/jni/

[40] Kerrisk, M. (2017) bzero(3) — Linux manual page. https://man7.org/linux/man-pages/man3/bzero.3.html.

[41] Albrecht, M. R., & Paterson, K. G. (2016, May). Lucky microseconds: A timing attack on amazon's s2n implementation of TLS. In Annual International Conference on the Theory and Applications of Cryptographic Techniques (pp. 622-643). Springer, Berlin, Heidelberg.

[42] Postel, J. (1981). Internet control message protocol (No. rfc792).

[43] Baker, F. (Ed.). (1995). Rfc1812: Requirements for ip version 4 routers.

[44] Detal, G., Hesmans, B., Bonaventure, O., Vanaubel, Y., & Donnet, B. (2013, October). Revealing middlebox interference with tracebox. In Proceedings of the 2013 conference on Internet measurement conference (pp. 1-8).

[45] Thirion, V., Edeline, K., & Donnet, B. (2015, April). Tracking middleboxes in the mobile world with traceboxandroid. In International Workshop on Traffic Monitoring and Analysis (pp. 79-91). Springer, Cham.

[46] Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., & Bowman, M. (2003). Planetlab: an overlay testbed for broad-coverage services. ACM SIGCOMM Computer Communication Review, 33(3), 3-12.

[47] Heffernan, A. (1998). Protection of BGP sessions via the TCP MD5 signature option. RFC 2385, August.

[48] Srisuresh, P., & Holdrege, M. (1999). IP network address translator (NAT) terminology and considerations.

[49] Jadin, M., Tilmans, O., Mawait, M., & Bonaventure, O. (2020). Educational Virtual Routing Labs with IPMininet. In ACM SIGCOMM Education Workshop.

[50] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., & Weiss, W. (1998). An architecture for differentiated services.

[51] Wikipedia contributors. (2022, January 30). Token bucket. In Wikipedia, The Free Encyclopedia. Retrieved 09:29, May 30, 2022, from https://en.wikipedia.org/w/index.php?title=Token_bucket&oldid=1068872165

[52] Kent, S., & Seo, K. (2005). Security architecture for the internet protocol (No. rfc4301).

[53] Piraux, M., & Bonaventure, O. (2020). Tunneling Internet protocols inside QUIC. Internet-Draft draft-piraux-quic-tunnel-01. IETF Secretariat. https://tools. ietf. org/html/draft-piraux-quic-tunnel-01.

[54] De Coninck, Q., & Bonaventure, O. (2017, November). Multipath quic: Design and evaluation. In Proceedings of the 13th international conference on emerging networking experiments and technologies (pp. 160-166).

[55] Hinden, R., & Deering, S. (1998). IP version 6 addressing architecture (No. rfc2373).

[56] Arapakis, I., Bai, X., & Cambazoglu, B. B. (2014, July). Impact of response latency on user behavior in web search. In Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval (pp. 103-112).

[57] SPDY whitepaper. http://www.chromium. org/spdy/spdy-whitepaper.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**