UNIVERSITE CATHOLIQUE DE LOUVAIN

ECOLE POLYTECHNIQUE DE LOUVAIN

MULTIPATH DTLS: DESIGN AND IMPLEMENTATION

Supervisors: Pr. Olivier BONAVENTURE
Pr. Olivier PEREIRA
Reader: Florentin ROCHET

Thesis submitted for the Master's degree
in computer science and engineering
option: Networking and Security

by Quentin DEVOS
Loïc FORTEMPS DE LONEUX

Louvain-la-Neuve
Academic year 2014-2015- June

We would like to thank our two supervisors, Olivier Bonaventure and Olivier Pereira, for their availability and their useful advices during the year,

Benjamin Hesmans who has provided a nice framework for our experiments,

Camille for her careful read-through of this thesis,

Sophie and Hélène for their support and advices all along this year,

Everyone who has help in a way or another to make this work possible.

Special mention to the Clinch Team to maintain the good mood in the INGI rooms.

Abstract

The goal of this master's thesis is to propose an extension for Datagram Transport Layer Security (DTLS) which is a protocol to communicate securely over an unreliable channel. Our extension, called Multipath DTLS, enables a DTLS session to use interfaces concurrently.

This feature becomes more useful with the emergence of devices with 2 interfaces or more (for instance LAN, Wi-Fi, 4G, ...). Smartphones, tablets and even notebooks are all good candidates for multipath protocols. The ability to connect or disconnect interfaces on the fly and seamlessly for the application was a dream becoming now reality.

Unlike other existing multipath protocols, the data security is a major objective and the design has been elaborated to keep the same security level as normal DTLS sessions.

We review the current state of the art by presenting DTLS in detail but also MPRTP. With some concepts borrowed from the latter, we present our design for Multipath DTLS, detailing all the modifications brought to the former protocol. In addition, we have implemented our extension inside an existing DTLS library and we disclose here some promising results.

This proposal might be used by a large amount of applications in very different contexts and on a substantial variety of devices.

Contents

List of Figures

List of Listings

Introduction

The DTLS provides security on top of datagrams protocols such as UDP. It was originally presented as an adaptation of TLS [1] for unreliable communications by Modadugu & Rescorla in 2004 [2]. This new standard arised whereas some software developers already had built their own way of dealing with secure communications over UDP. However, the interest for this protocol is increasing and its integration in existing systems when security is needed is more and more considered [3].

The main advantage of this protocol is its tolerance to frequent losses, which could be a requirement in some environments. This may also be helpful for real-time communication such as live streaming, where retransmission is not required. We will present later on all the potential applications that could benefit from using DTLS. Moreover, several existing commercial applications are already actively using it, like Cisco AnyConnect VPN[4].

Our objective in this master thesis is to assess the possibility to bring multipath ability to this protocol. Namely, we want to make it use multiple interfaces concurrently for the same DTLS session. This aspect has already been studied for other protocols such as MPTCP [5] or MPRTP [6]. Nevertheless,we must keep in mind the additional security requirement and guarantee at least the same level of security as standard DTLS.

In the first chapter of this master thesis, we present additional details about DTLS with a short summary of the foundations inherited from TLS. We give an example of how a typical handshake will take place and how security is guaranteed. Then, we present an overview of application data exchange and how losses are handled. Finally, we review the security considerations of DTLS.

Chapter 2 gives an insight into an existing multipath protocol, MPRTP. This is probably the closest protocol we have to be inspired of. Indeed, it is most of time used with UDP and has also to deal with losses and reordering. This chapter concludes the first part dedicated to the state of the art.

The following chapters constitute the second part of this thesis and focus on our proposition : MPDTLS. Chapter 3 details the different design aspects we have investigated, the various mechanisms built to make the multipath possible such as addresses advertisements and flow establishment. In addition, we review all the new packets introduced to support those mechanisms.

Chapter 4 describes our implementation of the design in an existing DTLS library. We show how the library works and the typical data flow. Then, we expose our data structures and processes to support multipath. Finally, Chapter 5 presents concrete results we have obtained with our implementation and a simple VPN application. In this part, we show that our design and the subsequent implementation are giving good results and really take advantage of multiple interfaces.

Part I

State of the Art

1 | DTLS

1.1 Overview

DTLS is a protocol designed to communicate securely over UDP. First presented in [2] by Modadugu and Rescorla as an adaptation of TLS for delay sensitive applications, the objective was to be as close as possible from TLS while preserving the same security level when working over an unreliable transport protocol. Like TLS, this protocol operates at level 5 in the OSI model (session layer). Therefore, the operating system does not provide any support for the protocol and every application has to manage it by itself. Typically, it implies either reinventing the wheel or using an existing library such as OpenSSL[7].

The current version of this protocol is DTLS 1.2 described in RFC6347 [8]. Our work is based on this version because no effort concerning a possible version 1.3 has been recently noticed. We may expect such a version will come after the discussions on TLS 1.3 that are taking place as we are writing these lines. For now, the suggested modifications have no impact on our current design as we explain in Section 3.4.

Even if this protocol is standardized for some years, only few important applications are supporting it. In the open source world, OpenVPN has planned to use it when UDP is selected. Unfortunately, it seems not to be a top priority and they are still dealing with a homemade protocol to do TLS over UDP. For commercial applications, we have only heard about Cisco AnyConnect [4]. It is a VPN client to be used with Cisco servers. Although the sources of AnyConnect are not available, an open source project is born from this application: OpenConnect[9].

Nevertheless, some people are working actively to make DTLS a default transport protocol when dealing with application level communications. More details about potential use cases are given in Section 1.4.

1.1.1 Objectives

Like TLS, DTLS has 3 strong requirements: authenticity, integrity and confidentiality.

Authenticity

We speak about authenticity when one host is able to verify that the messages are coming from another known host. Therefore, someone cannot pretend he is the sender of packets as the receiving host will detect it.

Integrity

The integrity is guaranteed if the information carried cannot be altered without being detected. As a consequence, someone between the communicating hosts cannot add, remove or modify a single bit without the hosts finding it out.

Confidentiality

Confidentiality assures that a non-authorized entity will not be able to extract an understandable content from a message. Thus, it is not sufficient to capture a conversation to get the information.

1.2 Foundations

In this section we present the way of working of DTLS and its roots. We will start with a reminder of TLS goals and principles since it is strongly related. Then, we will go through the major differences between these two protocols. Finally, we will list the steps of a typical connection and present the different messages involved.

1.2.1 TLS

TLS is a protocol designed to provide a secure connection over reliable transport. It takes place between a server and a client.

Once the channel is established :

- The client has *authenticated* the server (i.e. the server is really the one it pretends to be[1]) and optionally vice-versa.
- All messages can only be read by the other host (an eavesdropping is possible but the clear text cannot be obtained) thus achieving *confidentiality*.
- The packets cannot be replayed or modified on the line without the host finding it out thus assuring *integrity*.

To achieve these points, cryptography is used between both hosts and the packets are encrypted and authenticated. TLS authentication typically uses cryptographic signature based on public key together with a certification authority. During the handshake, the two hosts are negotiating which version of the protocol will be used together with algorithms to encrypt and authenticate the future messages. Many

[1] Under the assumption that the certificates can be verified by a reliable certification authority. It is the role of the Public Key Infrastructure (PKI) to provide and verify the certificates.

algorithms can be used to exchange the keys over an insecure channel. The two most common algorithms for key exchange are RSA and Diffie-Hellman.

The first one suffers from a lack of the perfect forward secrecy property. As stated in [10], *the perfect forward secrecy is guaranteed if the disclosure of long-term secret keying material does not compromise the secrecy of the exchanged keys from earlier runs*. In the case of RSA, if an attacker has recorded entirely some sessions and then discovers the server's private key, she is able to determine the symmetric keys used for every session. Moreover, in the discussions about the incoming TLS 1.3 standard, an agreement was reached to remove RSA key transport in this version[2].

An alternative to RSA is the Diffie-Hellman (DH) algorithm for key exchange described in [11]. The original approach used large prime numbers and the modular arithmetic to generate the master key. Nowadays, it is more common to use the elliptic curve approach since it allows to generate keys of smaller size than standard approach, thus increasing the performances[12]. In addition, to achieve perfect forward secrecy, the algorithms deal with ephemeral keys meaning that the DH parameters are different for every session. This leads to two well known algorithms : DHE (Diffie-Hellman ephemeral) and ECDHE (Elliptic Curve Diffie-Hellman ephemeral). Note there is also a "static" DH which does not change the parameters between sessions. This latter is considered less secured and is even removed in TLS 1.3 [13].

In addition to the key exchange algorithm, a choice must also be made concerning a second algorithm to sign the content and ensure the authenticity of the packets. Even if it is not recommended for key exchange, RSA can still be used to provide those digital signatures. Indeed, they are only needed during a short moment (a simple check when the packet is received) and will not reveal information about the content itself if the keys are compromised. Therefore, the forward secrecy property does not need to be guaranteed. Other algorithms may be used for digital signature such as ECDSA [14] which uses elliptic curves to sign content.

From this point, we will consider ECDHE for key exchange and RSA for signatures. Figure 1.1 illustrates TLS handshake with this configuration. The figure is taken from Cloudfare, a CDN provider. The visitor is the client and Cloudfare could be any TLS ECDHE compatible server.

In the first phase, the client gives its version together with a list of algorithms for encryption and authentication called cipher suites. The client also gives to the server a random token to be used later. The server then chooses among the ones proposed: the version and the cipher suite for the rest of the communication. A random server token is also included in the reply. In addition, the server may provide a session ID which allowing session resumption.

In a separated message, the server also provides its certificate to the client. It contains enough information for the latter to be able to verify server's identity. The certificate is generally emitted by a certification authority which is known by the client.

The next step comes with the server providing its DH parameters. In the case of ephemeral keys, the DH parameters must be different for every session. A digital signature using RSA with the server's private key is also included. The signature is computed not only on the information carried in this packet but

[2] `https://www.ietf.org/mail-archive/web/tls/current/msg12266.html`

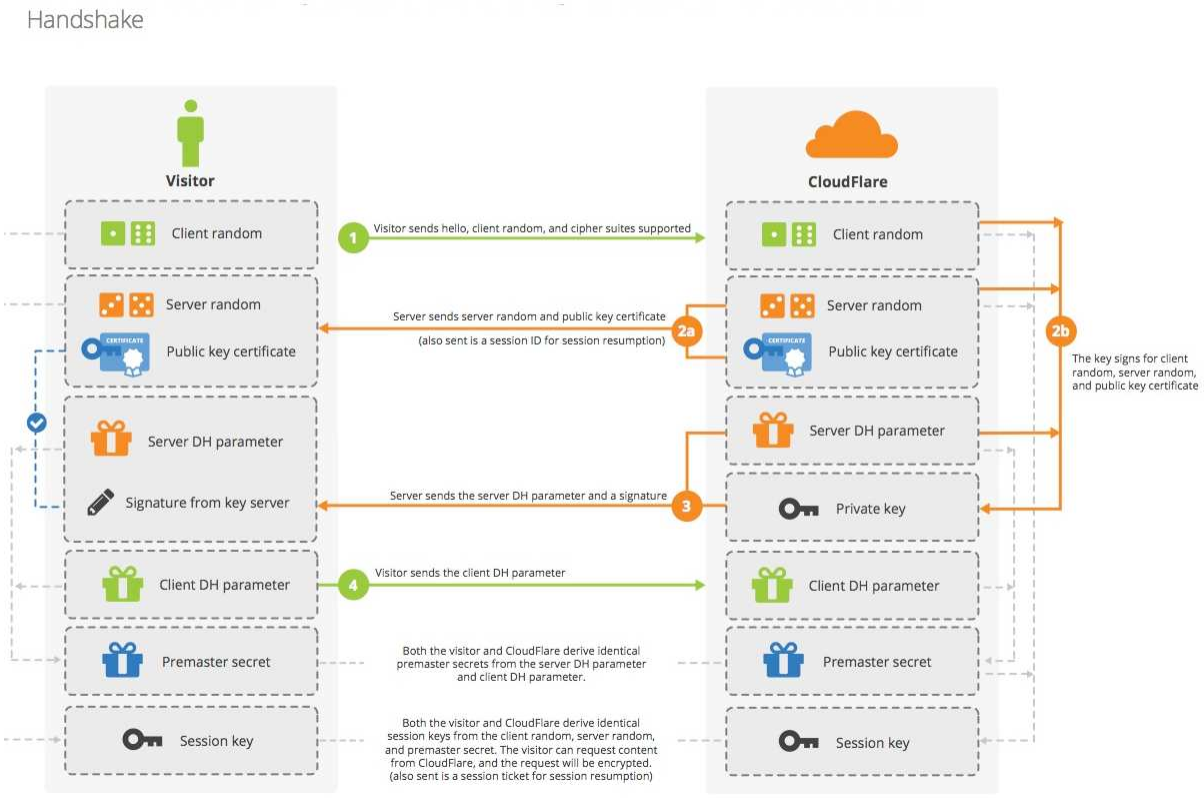

Figure 1.1: TLS handshake with (EC)DHE for key exchange (image from [15])

also on all the previous exchanged messages. This is needed to prevent a man-in-the-middle attack. The genuine server is supposed to be the only one to hold the private key.

Then, the client provides its own DH parameters. Note that no signature is provided in this case as it is only optional to authenticate the client. Indeed, in some applications such as websites, this is not a requirement. After this step, both client and server can deduce the *PreMasterSecret* and thus the master key with the random tokens exchanged before.

All messages following the handshake (containing application data) will be encrypted with symmetric cryptography using the keys deduced from the master key for this particular session. This will ensure the confidentiality. The two other objectives, authenticity and integrity of the messages, are ensured by a keyed-hash message authentication code (HMAC). It is generated as a hash on the content of the packet but the hash algorithm also integrates the secret key. Any cryptographic hash function can be used for this purpose but it is recommended to use at least `SHA256`. The choice of this function is also negotiated during the client hello, server hello phase. Even if it is better to first encrypt and then MAC the output as stated in [16], TLS works in reverse order for historical reasons. Currently people think it was a very bad idea from a security point of view and a RFC [17] was published last year to push for the adoption of the encrypt-then-MAC technique on existing systems. It will probably be part of the TLS 1.3 RFC [13] to switch definitely to this better practice.

To prevent malicious replays from disturbing the normal communication, TLS authenticates each packet with the associated sequence number. The sequence number is incremented for every packet sent but separately for each direction. Therefore each host has two counters: one for the sent packets and a second one for the received packets.

Thanks to a reliable transport protocol, these counters must never be transmitted explicitly because each host can compute them. As a consequence, if an attacker tries to replay packets, they will simply be rejected because the HMAC will not match with the most recent sequence number.

1.2.2 Differences between TLS and DTLS

Unreliability

The unreliability brought by UDP causes problems if we apply TLS over datagrams without any modification.

As we explain in the previous section, TLS uses implicit sequence numbers to authenticate packets. If a loss or a reordering occurs, then the sequence number considered by the receiving host will be wrong and the integrity check will fail. To solve this problem, DTLS modifies the record layer and adds a new field to carry the sequence number in every packet. The DTLS 1.2 record layer is presented on Listing 1.1.

```
struct {
   ContentType type;
   ProtocolVersion version;
   uint16 epoch;                                  // New field
   uint48 sequence_number;                        // New field
   uint16 length;
   opaque fragment[DTLSPlaintext.length];
 } DTLSPlaintext;
```

Listing 1.1: DTLS record layer

Epoch is used by endpoints to identify the keys used to protect the payload. The epoch is incremented by one once a `changeCipherSpec` is sent. Initially, the epoch is set to 0 for the handshake and the following `changeCipherSpec` will be the first message being part of the next epoch. The sequence number is set back to 0 once a new epoch has started. Due to the risks of retransmission or reordering, the endpoint need a way to know which set of keys were used to crypt the payload, because it cannot simply assume it is the latest negotiated keys. This requirement is fulfilled thanks to the epoch field.

Another problem raised by unreliability is the fact that in TLS the handshake has to follow a particular order and if not the connection is simply aborted. This is clearly incompatible with losses and retransmissions. To handle potential losses in the handshake phase, implementation must retransmit the last packet after a given time if no answer has been received from the other host (see [8] Section 4.2.4).

To deal with possible reordering, all handshake messages have a given sequence number. So if a message is received before another one, the first message must be queued for further processing.

Messages size

The message size is a potential issue because TLS (and thus DTLS) messages can be as large as $2^{24} - 1$ bytes while UDP datagrams are often limited to 1500 bytes. Therefore, each DTLS handshake message may be fragmented over several DTLS records and then rebuilt. One fragment must fit in a single IP datagram to avoid IP fragmentation.

The benefits to this requirement are stated in the original DTLS paper [2] :

- The DTLS layer does not need to buffer partial records, host memory can be used more efficiently.
- It is quite possible that datagrams carrying the remaining record fragments are lost, in which case the received fragments are useless and cannot be processed.
- Buffering record fragments would unnecessarily complicate a DTLS implementation without providing any obvious benefit.

Note that even if IP fragmentation occurs, DTLS will still operate correctly on top of re-assembly since it is transparently handled by the kernel.

The TLS handshake message is modified as shown on Listing 1.2. The handshake message following this structure is also encapsulated into the record layer presented on Listing 1.1.

```
struct {
  HandshakeType msg_type;
  uint24 length;
  uint16 message_seq;                              // New field
  uint24 fragment_offset;                          // New field
  uint24 fragment_length;                          // New field
  select (HandshakeType) {
    case hello_request: HelloRequest;
    case client_hello:  ClientHello;
    case hello_verify_request: HelloVerifyRequest; // New type
    case server_hello:  ServerHello;
    case certificate:Certificate;
    case server_key_exchange: ServerKeyExchange;
    case certificate_request: CertificateRequest;
    case server_hello_done:ServerHelloDone;
    case certificate_verify:  CertificateVerify;
    case client_key_exchange: ClientKeyExchange;
    case finished: Finished;
  } body;
} Handshake;
```

Listing 1.2: DTLS handshake message

Anti-Replay strategy

In TLS, sequence numbers follow sequentially and a packet cannot be replayed because the HMAC verification will fail immediately. Such a drastic solution cannot be applied in DTLS because reordering or losses may occur. The DTLS 1.2 RFC [8] sec. 4.1.2.6 describes the procedure to avoid replay using a sliding window. This strategy considers packets as valid even if they have been delayed or reordered while they are relatively close from the latest packets received. At the same time, the implementation must remember the packets received inside the sliding window and silently discard the replayed packets. Therefore, the MAC verification will only be done if the packets have not been received before, otherwise an undesirable packet is discarded at the record layer stage.

Anti-DoS for the handshake

In order to prevent DoS (Denial of Service) attacks, DTLS adds an additional step in comparison with the TLS handshake. Indeed, an attacker could send a lot of `ClientHello` messages and the server would have to keep a state for each of them, rapidly increasing the memory needed. Instead, the first `ClientHello` will not directly trigger a `ServerHello` but a `HelloVerifyRequest`. This message only carries the protocol version and a cookie as presented in Listing 1.3. This cookie must be retransmitted in the next *ClientHello* message. As we can seen on Listing 1.4, a new field has effectively been added to the `ClientHello` message.

```
struct {
 ProtocolVersion server_version;
 opaque cookie<0..2^8-1>;
} HelloVerifyRequest;
```

Listing 1.3: DTLS HelloVerifyRequest message

```
struct {
 ProtocolVersion client_version;
 Random random;
 SessionID session_id;
 opaque cookie<0..2^8-1>;                              // New field
 CipherSuite cipher_suites<2..2^16-1>;
 CompressionMethod compression_methods<1..2^8-1>;
} ClientHello;
```

Listing 1.4: DTLS ClientHello adapted message

The additional `HelloVerifyRequest` message is also a counter measure against amplification attacks. Such attacks are trying to redirect traffic to overload a particular target by using another server to generate bigger messages. These attacks are possible if the generated message is larger than the one triggering it. It is the case with DNS since the reply is larger than the request if a particular domain has multiple DNS records. It becomes then a vector for DoS attacks (see Figure 1.2 taken from [18]).

For DTLS, the `ServerHello` is not larger than the `ClientHello` but amplification attack could still be possible (without the `HelloVerifyRequest`). Indeed, other messages are sent together with the

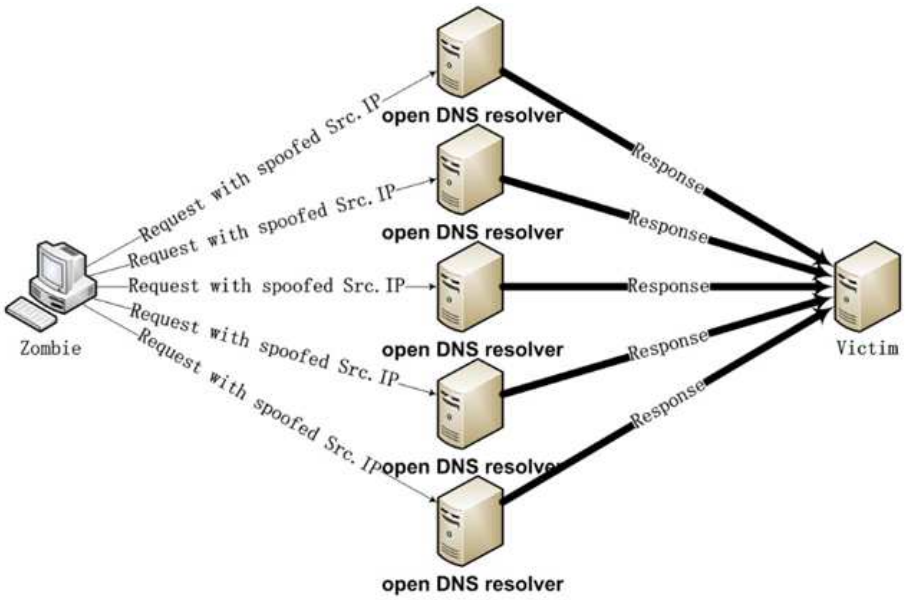

Figure 1.2: Example of amplification attack with DNS

`ServerHello`, including the one carrying the certificate (that could be quite large). Therefore, the additional step introduced with the `HelloVerifyRequest` (which is relatively small) prevents this kind of attack because the client has to prove it can answer at this address.

When the client sends its first `ClientHello`, the cookie field is initially empty. The server will then generate a new cookie which allows for a stateless exchange. It sends the `HelloVerifyRequest` with the cookie and doesn't remember anything about the client[3]. The DTLS server must generate cookies in such a way that they can be verified without retaining any per-client state on the server. A way to compute it is given in the RFC6347 section 4.2 [8]:

```
Cookie = HMAC(Secret, Client-IP, Client-Parameters)
```

The `Secret` is random and generated by the server. It could be periodically refreshed to invalidate previous cookies. Once a `ClientHello` has been received with a cookie, the server recomputes it and if both match, the connection can continue as in TLS with a `ServerHello` message.

1.3 Typical DTLS communication

1.3.1 Handshake

Figure 1.3 depicts a typical handshake with authentication of the server only and the ECDHE-RSA cipher suite[4]. All messages before the `ChangeCipherSpec` are part of epoch 0. Their role is to negotiate the algorithms that will be used to compute the master key. The `Client Hello` has also the role to carry

[3]This is indeed an exception to the retransmission measures. A `HelloVerifyRequest` will never be retransmitted, the client must first send another `ClientHello`.

[4]It means that Elliptic Curve Diffie-Hellman ephemeral is used for the key exchange and RSA for the signature.

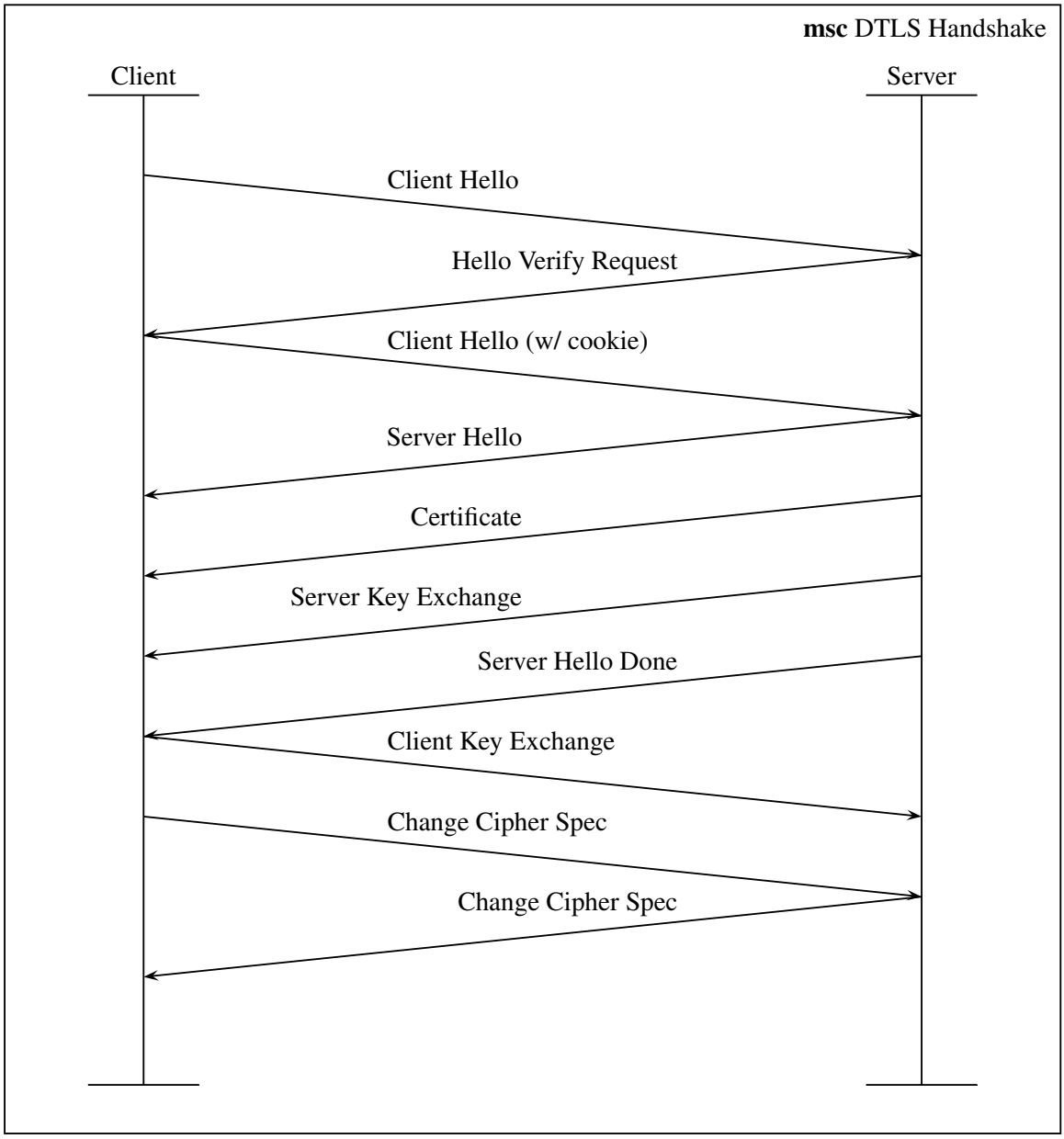

Figure 1.3: DTLS handshake with ECDHE-RSA cipher suite

potential extensions supported by the client. If the server supports them as well, it will include them in the following `Server Hello` message. This is also where we chose to advertise our MPDTLS extension as explained in Chapter 3.

The only difference with TLS (Section 1.2.1) is the presence of a second `ClientHello`. The first one is sent with no cookie and the second one must repeat the cookie received with the `HelloVerifyRequest`. Then the server confirms the cipher suite to be used through the `ServerHello` and directly sends its certificate and its ephemeral public key[5] including a signature of all previous exchanged messages. Finally the server sends the `ServerHelloDone` to explicitly say that the handshake is finished on its side.

[5]In the case of ECDHE, it consists of the public DH parameters. Namely the domain parameters to define a particular curve.

Thereafter the client must also send its DH ephemeral public key. At this point, both parties have all the elements in their hands to compute the premaster secret. Note that unlike RSA, the secret is never communicated through the channel. With the random information exchanged through the `ClientHello` and `ServerHello`, each host can also compute the master key on its side. Consequently each peer will use this key to derive the set of keys used to communicate in the next epoch. The first messages of epoch 1 are the `ChangeCipherSpec` messages. They explicitly mark the separation between the handshake and the rest of the communication. Even if they are technically separated from the handshake, they are still considered as being part of it from a conceptual point of view.

1.3.2 Application data

Once the handshake has been completed, the peers can exchange Application Data packets. They are first encrypted and authenticated using the parameters negotiated earlier, then encapsulated into a Record Layer (Listing 1.1).

DTLS doesn't bring any additional guarantee concerning the reliability of the link. As shown in Figure 1.4, packets may be lost on the way and the application must deal with it. The sequence number for each message is indicated between chevrons. Sequence numbers are carried in clear inside the record layer, so each host knows how to verify the MAC and authenticate the packet.

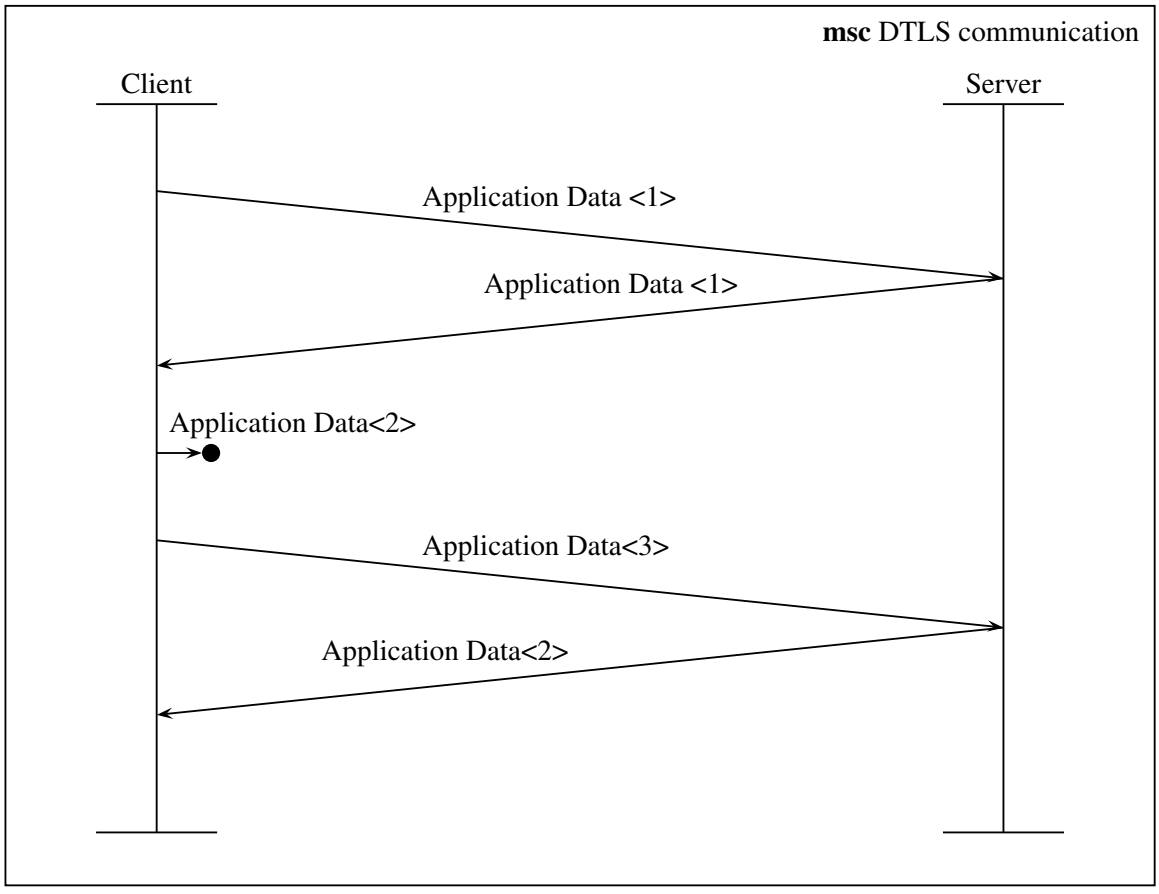

Figure 1.4: DTLS application data exchange

1.4 Use cases

DTLS can be used almost everywhere UDP is used. We can think about applications such as VoIP (Voice over IP), multimedia, online gaming. Every application that wants to secure its communication but still benefit from faster transmission time of UDP may take advantage of using DTLS.

Of course the application has to cope with losses or reordering, but this is already the case for real time communication. As regards telephony, blank sounds replace the missing packets. For online gaming, high speed communication between the server and the client is needed to determine the character's position for instance. But if one packet is lost, the position can be determined by the next packet and the application will not be disturbed.

For video streaming applications, techniques are used to introduce a slight redundancy at the codec level in such a way that if some packets are lost, one frame may be missing but at least the video is still watchable[6]. It is also possible to introduce a small buffer to handle reordering.

In a recent Internet draft [3], companies are trying to make DTLS the default sub-transport protocol for all application-level protocols when security is needed. This will raise DTLS to the rank of "good practice" for secured communications.

Another important use case for DTLS comes with its integration in WebRTC[19]. This is an open project which has an objective to bring real time communications to browsers and mobile applications via simple APIs. In this context, DTLS has been chosen to ensure the communication between two peers (i.e. browsers or mobile apps). The internet draft describing the security architecture of WebRTC [20] proposes to use SRTP over DTLS for multimedia communications and DTLS alone for any other kind of data. Figure 1.5 is taken from a presentation [21] made by E.Rescorla and shows how the connection is set up in this proposed design.

By the peer to peer nature of this application, a signalling server is needed to act as a "rendez-vous" point and to perform NAT traversal. Then the handshake can take place and identity providers issue and verify certificates. Finally all data are transmitted through the DTLS transmission and multiple SRTP sessions can use the same DTLS session.

To sum up, all these applications may benefit from using DTLS but also MPDTLS as it will allow more resilience and better performances. We will demonstrate this point in the following chapters.

1.5 Security considerations

Most of the security considerations are the same as those of TLS 1.2 [1] since DTLS is only an adaptation of TLS for unreliable transport protocols.

The attacks known for TLS could theoretically be used for DTLS as well. This is the case for the "secure

[6] Of course, the application must a use a proper codec and compression method.

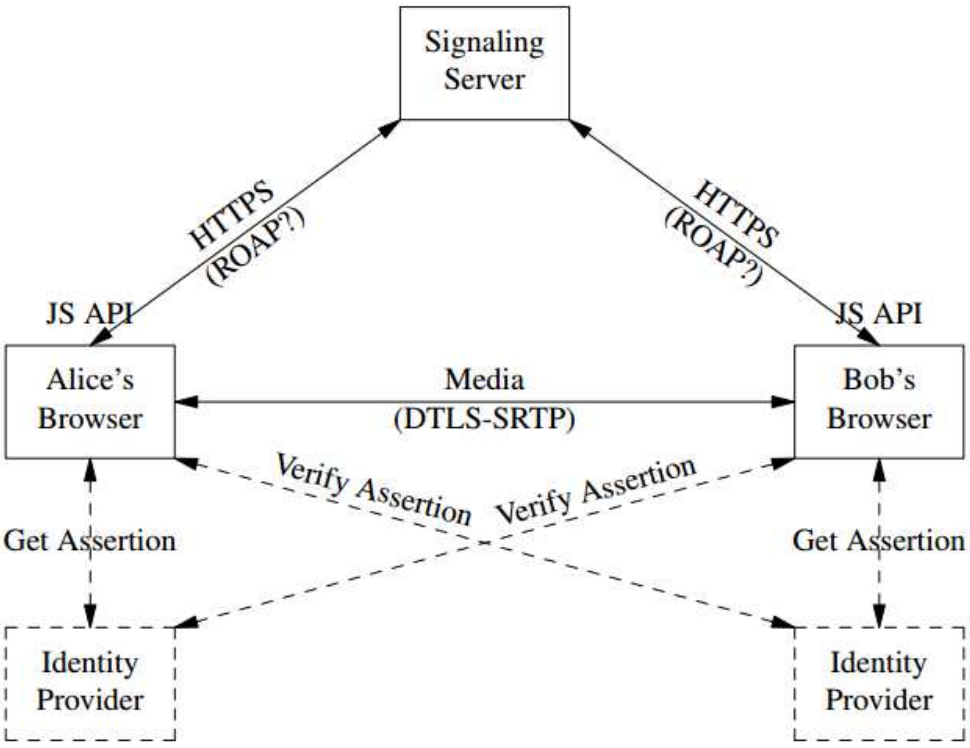

Figure 1.5: Proposed design for WebRTC security architecture

renegotiation". As a reminder, this vulnerability allows an attacker who can hijack a HTTPS connection to add custom request to a communication between the client and the web server. Even if the attacker cannot retrieve the content of the communication, it can still have dramatic consequences. As depicted on Figure 1.6 taken from [22], an attacker can inject bank orders and then trigger the renegotiation with a real client. The bank order is only delayed and when the renegotiation is completed, it is executed before anything else. The best solution up to know is to disable by default this feature or to implement the extension described in [23].

Other attacks could also be applied to DTLS and be more efficient on DTLS than on TLS, namely the DoS attacks. Two main categories can be identified :

- The blind DoS : packets are being transmitted with a spoofed IP address to redirect the traffic to another target (amplification attack) or to simply not diclose the IP address of the attacker. They are called blind because the attacker will never receive any answer from the server. They are made easier with UDP because no handshake is needed with the server.
- Computational DoS : the attacker can reply to certain messages but will try to optimize the ratio between the work he has to do and the work he is asking to the server.

Blind DoS are made impossible in TLS because the attacker must first complete the TCP handshake and thus prove he can answer at this address. As presented earlier, the introduction of the `Hello Verify Request` will prevent this kind of attack for DTLS too. Although it is not a strict requirement to

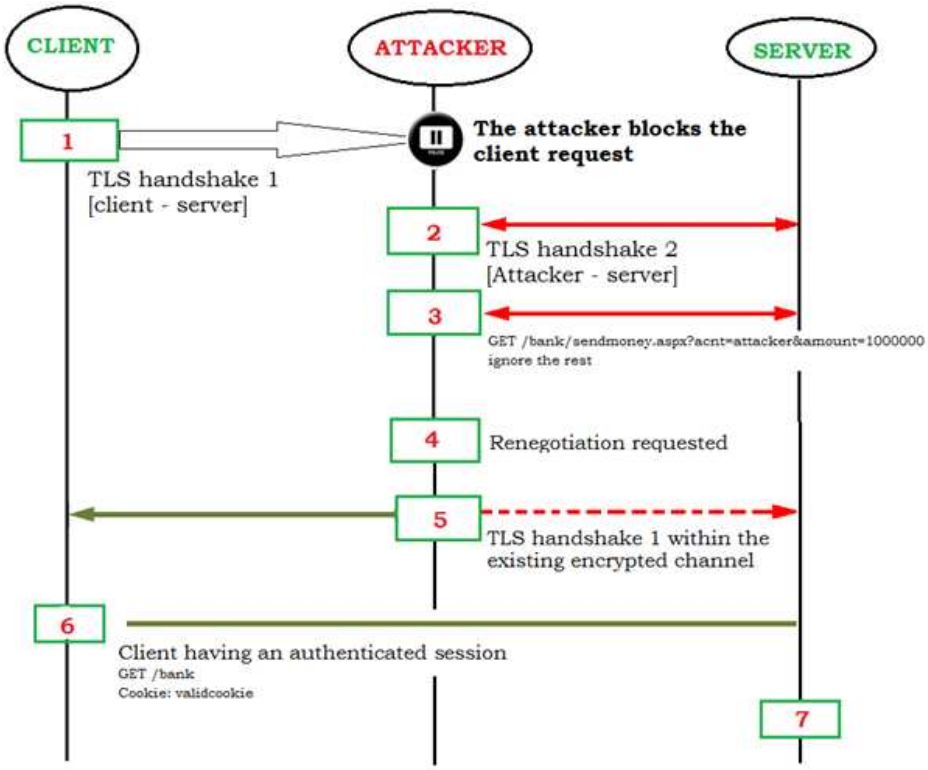

Figure 1.6: TLS renegotiation vulnerability

implement this feature in DTLS ([8] Section 5), it is strongly recommended for DTLS servers unless there is a good reason to think that no amplification is possible in their environment.

Nevertheless the second type can still be used to disturb a server running DTLS or even TLS. As explained by E. Rescorla in [24], nothing is done at the design level to prevent these kind of attacks because it would imply putting a lot of work on the client side. It is generally a bad idea for performances. Moreover, the attackers are often using infected computers of casual people to launch DoS attacks. Therefore they have a lot of CPU power available anyway.

1.6 Heartbeat extension

The heartbeat extension is unfortunately known for the famous heartbleed bug[7] detected in April 2014. Every server using OpenSSL at this time was vulnerable and part of the memory could be retrieved (including private keys, passwords...). It is important to note that the error came from the implementation of the extension inside OpenSSL and not from the standard itself.

Anyway, this extension can be really useful to assess the availability of a link and also provides a keep-alive feature. It is quite simple but it also allows for extensibility and new messages as stated in RFC6520 [25]. The structure of a heartbeat message is presented on Listing 1.5. This will take place on top of the Record Layer (Listing 1.1). The advertisement of this extension is made through a Hello extension. It

[7] http://heartbleed.com/

also defines the behavior of the hosts upon the reception of Heartbeat messages.

```
struct {
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[HeartbeatMessage.payload_length];
  opaque padding[padding_length];
} HeartbeatMessage;
```

Listing 1.5: Heartbeat message

For now, only two types of Heartbeat messages are in use : `Heartbeat Request` and `Heartbeat Response`. The response must contain the same payload as the one in the request triggering it. A simple exchange is presented on Figure 1.7.

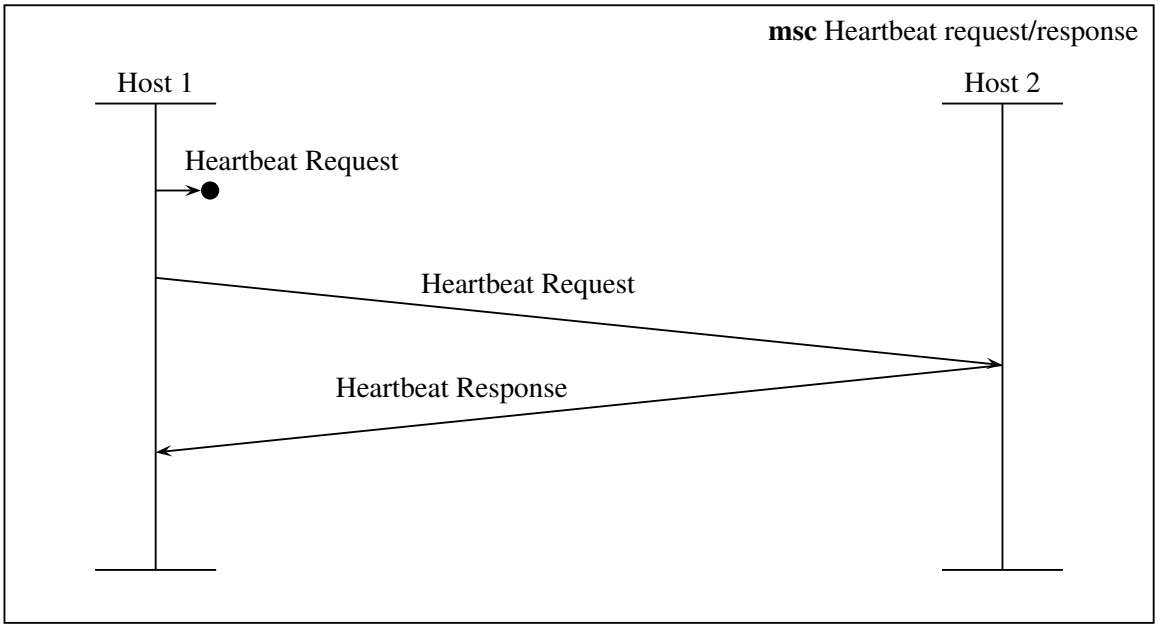

Figure 1.7: Heartbeat requests and responses

Other messages may be implemented since a new IANA section has been opened to register the different `HeartbeatMessageType`. Indeed, as you will see in the Chapter 4, we have designed a new type of message used to transmit a timestamp.

2 | MPRTP : Multipath over UDP

MPRTP (Multipath Real-time Transport Protocol) is an extension to RTP currently standardized in [6]. This protocol, described originally in [26], tries to bring multipath feature to the traditional RTP. The idea is to take advantage of the presence of multiple interfaces to improve the way information is transmitted.

Although other multipath protocols exist, such as MPTCP [5], we chose to study more in detail the case of MPRTP. Indeed, it is probably the closer existing protocol that operates over UDP and does multipath with addresses advertisement, link quality assessment, etc. However, MPRTP does not need cryptography because the protocol does not guarantee any authenticity or integrity of the messages.

In this chapter, we will give an overview of RTP, before going through the different challenges brought by the multipath aspect to see how MPRTP solves it.

2.1 RTP

The Real-time Transport Protocol (RTP) defined in `RFC3550` [27] is a protocol to deliver real-time information such as audio or video across the network. It is used by various applications : VoIP, video conferences, television services... All connections are unidirectional with a source and a destination. The same RFC proposes another protocol, RTCP (RTP Control Protocol), that offers a way to monitor a link by regularly providing feedback from the receiver to the sender. The feedback includes various parameters such as the loss rate, the RTT, etc. As stated by the RFC3550, RTP is used most of the time in conjunction with RTCP.

2.1.1 Connection Establishment

RTP does not provide any support to open a particular connection. Since it is used most of the time over UDP, a handshake is never established. Instead, RTP relies on out-of-band protocols to manage the session establishment. Although any protocol can potentially be used, SIP (Session Initiation Protocol) has become the most common way to open sessions for VoIP applications. Figure 2.1 presents a conceptual schema of how it takes place. An external server is generally used to negotiate the ports for the connection and can provide NAT traversal feature.

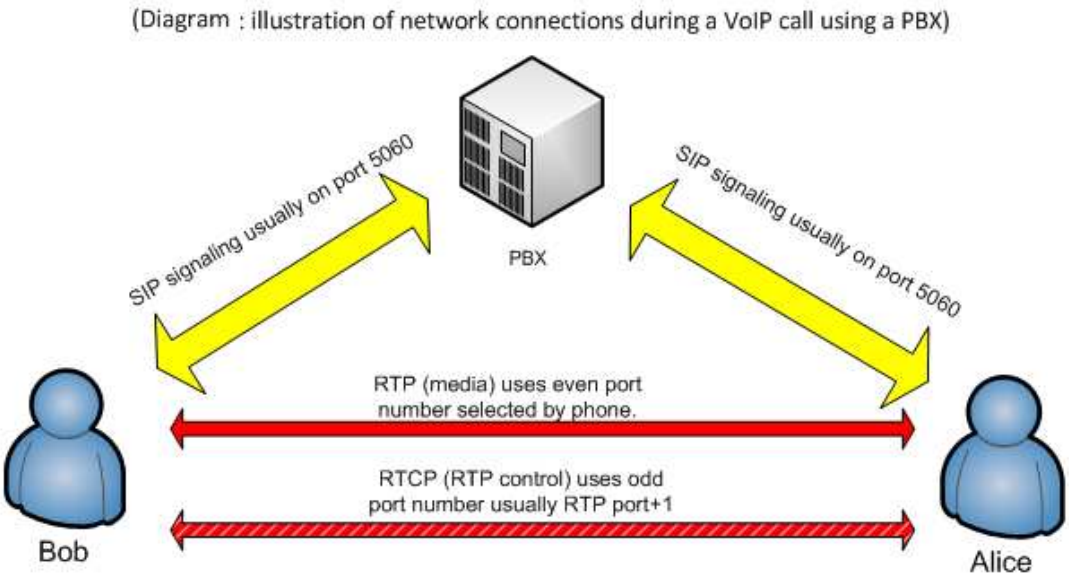

Figure 2.1: How an RTP connection takes place for VoIP (image from [28])

Two flows are initialized : one for RTP and the other for RTCP. They usually use consecutive ports while RTP must use an even port number ([27] Section 11). So only one pair of ports must be advertised for each host. This is only true for a unicast session. IP multicast can also be used with RTP and offers better performances if a large number of clients must access the same media resource. However, we won't give any details about how RTP multicast sessions are established since is it out of the scope of this thesis.

2.1.2 RTCP

RTP is just responsible for sending data to the receiver, with some additional information inside the packet compared to simple UDP. It includes sequence numbers, timestamps, information about the sources and control bits. But this mechanism alone does not give any hint about packet losses or the congestion of a link. This is why RTCP is essential in most of the applications.

According to [29], RTCP performs four tasks :

1. It provides feedback on the quality of the data distribution. It plays the role of a congestion control mechanism.

2. It carries an identifier for the RTP source called the canonical name. It allows each host to identify the participants in a particular flow. It can be useful when dealing with multicast.

3. It allows the source to compute the rate at which it has to send the packet. This takes into account the number of participants to the session.

4. It conveys minimal session control information, such as participant information to be displayed on user interface, for instance. This functionality is left optional.

To do so, it carries Sender and Receiver reports. These reports are transmitted to the other party to estimate the quality of the link. They include information such as:

- the packet count, octet count (received/sent)
- the cumulative number of packets lost and fraction lost
- the highest sequence number received
- different timestamps to estimate RTT and jitter.

2.2 MPRTP

Currently, RTP is not able to work "as-is" in multipath mode because it operates at the session level and not at the transport level. Therefore RTP deals only with a particular media flow and not with the traditionnal 5-tuple. The idea of MPRTP is to use multiple RTP flows concurrently on multiple interfaces. Figure 2.2 presents the system overview. As for RTP, the communication of the media is unidirectional : from a sender to at least one receiver.

The sender is in charge of gathering the paths characteristics and scheduling the packets accordingly. The receiver must reorder the media packets to present them as a continuous stream for the application. The codec can then rebuild the media as if it was coming from a normal RTP flow. We will explore in the following sections the different components needed to add this multipath feature.

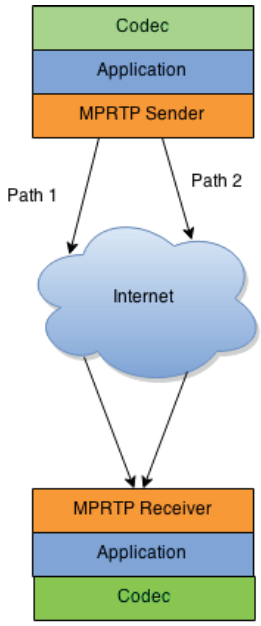

Figure 2.2: MPRTP overview

2.2.1 Design goals

Adapting to bandwidth changes

If a link is congested, the scheduler must reduce the number of packets on this link and re-distribute the load among less congested links. But a small quantity of traffic must still be sent through this link just to monitor the link's characteristics (RTT, jitter). A similar approach is used in Multipath TCP [30]. Of course, the changes of repartition among the flows must be smooth and absorb small perturbations. Especially when it is used with mobile phones, as the switch between antenna may introduce a loss of connection on a very short period of time.

Overcoming packet skew

When different paths are used over the Internet, different delays will likely be encountered as shown in [31]. As a consequence, the packets will arrive out of order. To mitigate this problem, a buffer large enough is needed at the receiver side. For multimedia communications, this is critical, as song or video has to be read in-order. However, the scheduler could take care of rigorously selecting the flow in order to minimize the skew.

Choosing transmission paths

Among the available paths, the MPRTP implementation must choose a subset of them to send the packets. This choice is actually an optimization issue based on multiple parameters. An endpoint may chose to optimize some of them. Following the scenario and the application needs, the scheduler may minimize losses, minimize latency or maximize end-to-end capacity.

2.2.2 Getting subflow information

To meet these goals, the scheduler will need to get information about the different subflows. Just by using the reports from sender and receiver as exposed in the Section 2.1.2, MPRTP already have an overview of the session as a whole. However to get per-subflow information, it has to introduce RTCP per-subflow reporting. In order to decouple the subflows from the session, an ID is assigned to each subflow. A subflow is considered as unique if the source and destination IP addresses and ports are different, which, together with the protocol, gives a 5-tuple.

To get an estimation of the packet jitter, packet loss and packet drop for a particular subflow, a subflow-specific sequence number is introduced. This new sequence number and the subflow ID are both carried in a header extension. This additional header is standardized in [27] and will therefore allow backward compatibility.

2.2.3 Addresses advertisement

To establish new paths, a mechanism must advertise the addresses to the other host. In MPRTP, the choice is currently [6] left to the application to use in-band or out-of-band communication to take care of it. "In-band" means we use the same protocol (in this case MPRTP) to exchange our addresses while "out-of-band" will imply another protocol that could be reliable for instance.

```
struct {
   byte MPRTCP_Type;
   byte block_length;
   uint16 RTP_port;
   byte InterfaceAddress[(block_length-1)*4];
 } MPRTCPInterfaceAddress;
```

Listing 2.1: MPRTCP Interface Advertisement

We focused on in-band communication since that is the way we choose working in our proposed design (see Section 3.1.2). The MPRTCP extension for interface advertisement is defined in [6] sec. 9.3. The structure used to carry one particular IP address and port is shown on Listing 2.1.

The `MPRTCP_Type` indicates the type of interface address. For now, 3 types are defined:

1. an IPv4 address
2. an IPv6 address
3. a DNS name

The `block_length` is the size of the block (i.e. the whole structure) in 32-bit words. For an IPv4 address it should be 2 (1 for the interface itself and 1 for the information before), for an IPv6 it should be 5 and for a DNS name it will be variable.

The `RTP_port` must be a valid RTP port number different from 0. The `InterfaceAddress` contains the actual IP address and its size is determined by the `block_length`.

Note that Listing 2.1 only presents the block to advertise one interface but such blocks will be combined in a unique packet to take into account multiple interfaces.

As stated in the working draft, all the interfaces in use must be transmitted. Moreover an endpoint must advertise its interfaces whenever an interface appears or disappears and when it receives an Interface Advertisement. Having to reply to an Interface Advertisement provides somehow an acknowledgment mechanism that could be useful in an unreliable environment.

2.2.4 Scheduling Algorithm

The scheduler is one of the most important aspect and a critical point if the multipath approach has to overcome the single path one. In the case of MPRTP, the sender must choose on which path to send the packets in order to guarantee a constant bit rate. A simple example with two paths is shown on Figure 2.3.

The goal of the scheduler is to compute the fractional distribution to distribute packets accordingly to paths characteristics. This means that a link with a smaller RTT will probably be more loaded than another one.

However, the packets may arrive out-of-order if a significant difference in terms of RTT is noticed between paths. To compensate this reordering, a dejitter buffer must be set up at the receiver's side (Figure 2.4). The buffer must be large enough to compensate variation in per-path RTT.

This buffer will rearrange the packets to put them in order and deliver the information to the application as a single stream.

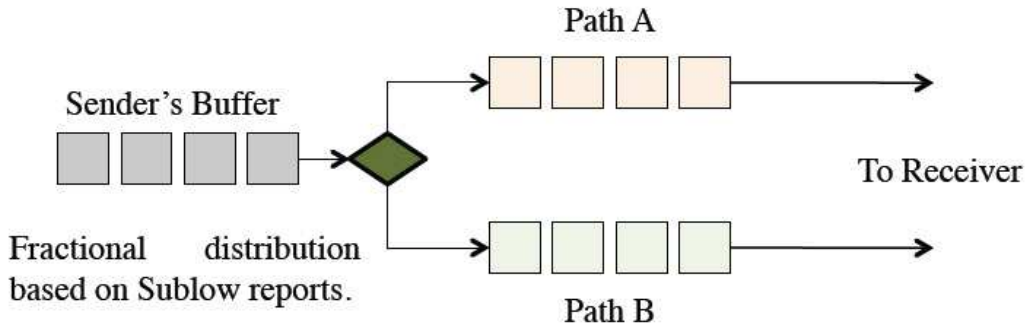

Figure 2.3: MPRTP Sender's Scheduler

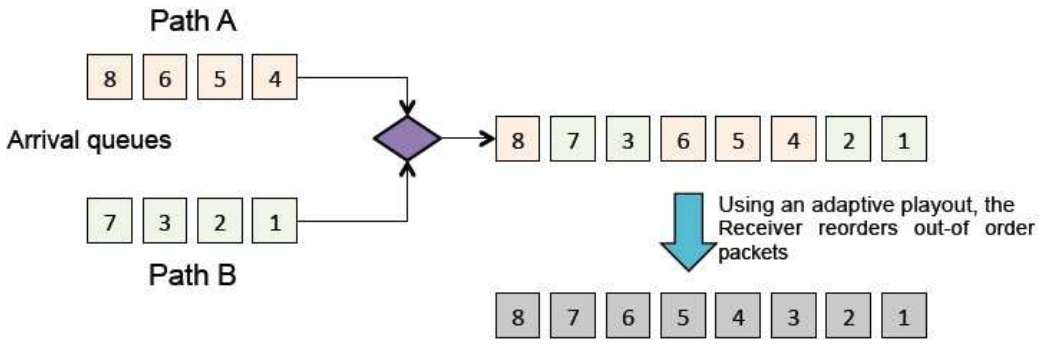

Figure 2.4: MPRTP buffer at receiver's side

Receiver rate

The receiver rate (RR) is computed for every subflow as the total number of bytes that have been transiting through the subflow between two consecutive receiver reports. This rate takes into account only the packets correctly received by considering the loss rate. When the receiver builds its i^{th} report, the RR is computed as follows :

$$RR = \frac{(\sum_{k=HSN_{i-1}}^{HSN_i} sizeof(X_k)) \times (1 - L_i)}{(t_i - t_{i-1})}$$

where HSN_i stands for the "Highest Sequence Number" received in the report i. So the sum actually represents the total number of bytes sent between two consecutive receiver reports. L_i is the loss rate reported for this period of time.

Calculating the fractional distribution

The computation of the fractional distribution is not done each time the sender receives a receiver report. An interval timer is instead defined to run it more or less frequently following the network information. This interval expressed in seconds is computed as follows :

$$SchInt = \lambda \times S_{interval}, \quad 0.5 \leq \lambda \leq 1.5$$
$$\lambda = 0.5 + Rand(0.0, 1.0)$$

The randomisation factor λ is used to prevent multiple senders using the same paths to launch the computation at the exact same time. This would lead to a delta in server load. $S_{interval}$ is a value between the minimum and maximum RTCP interval $[32; 39]$. It will be kept small at the beginning of the communication to recompute the distribution more frequently but progressively increases to the maximum. If congestion is detected, the $S_{interval}$ is reduced to recover quickly if the link becomes non-congested later on.

At the end of the interval, the distribution needs to take into account the state of each link in terms of congestion. For this purpose, three categories are defined : congested, mildly-congested and non-congested. To determine the category of one particular subflow, the rules defined in [32] are used. This Internet draft aims at providing concrete rules to determine if a RTP application must stop sending packets to avoid congestion.

The proposed algorithm to split the traffic among all available paths will take into account the degree of congestion and will reduce the load of congested links.

$$SB[i] = min\left(\frac{RR[i]}{\sum_m RR[m]} \times MR, \quad \alpha \times RR[i]\right) \qquad \text{if congested}$$
$$SB[i] = min\left(\frac{RR[i]}{\sum_r RR[r]} \times MR, \quad \beta \times RR[i]\right) \qquad \text{if mildly-congested}$$
$$SB[i] = \left(\frac{RR[i]}{\sum_w RR[w]} \times (MR - AP)\right) \qquad \text{if non-congested}$$

where $SB[i]$ is the sending bit rate for a particular path i, MR is the media rate (i.e. total bit rate) and AP is the allocated bit rate. The indexes m, r, w are used to represent the set of paths congested, mildly-congested and non-congested respectively. The minimum function is fixed here to limit the maximum usage of a congested link but will not change anything if the link is already underused. Of course, this should be repeated for every path i.

The parameters α and β must be tuned by experiment. In section 5.1 of [26], β is set to 0.8 and α to 0.5 after a series of tests in various configurations. These values lead the scheduler to reach the optimum fractional distribution quicker than other values.

Part II

Multipath DTLS

3 | Protocol Design

In this chapter, we will explore the different additions and modifications made to DTLS to support the Multipath capability. These changes are categorized in three groups:

- Advertising the extension and the interfaces
- Establishing secure sub-flows without introducing attacks vectors
- Gathering and exchanging statistics data about the health of each flow, regardless of the others.

In addition to the complete description provided here, all the changes made to DTLS are summarized in Appendix B.

3.1 Multipath advertisement

Our main purpose is to set up this protocol as an extension of DTLS. In this way, we can reuse as much as possible the principles established by Rescorla and Modadugu in [2]. This also explains why we tried not to change the existing DTLS frames and to add instead new frames for new usages.

3.1.1 Extension discovery

The first step, and the first requirement, was to remain compatible with the standard DTLS client and server. To do that, the MPDTLS extension discovery is made through a new entry in the extensions list of the `ClientHello` and `ServerHello` messages. If a MPDTLS-unable server receives a `ClientHello`, it will ignore the option as it is specified in the TLS 1.3 specifications ([13]). This option is carried as any other TLS Hello Extension (Section 7.3.2.5 from the same draft) with the following format:

```
struct {
  ExtensionType extension_type;
  byte          extension_length[2];
  byte          extension_data[1];
} Extension;

enum {
    mpdtls_extension(TBD, 42 in dev), (65535)
} ExtensionType;
```

Listing 3.1: MultiPath DTLS Extension structure

This extension is simply carrying a byte indicating if the host supports MPDTLS or not (so it carries `0x01` or `0x00` respectively).

After the exchange of the `HelloVerifyRequest` and `ClientHello` with Cookie, the server will send back a `ServerHello` containing the same extension if it wants to support MPDTLS features. Besides this MPDTLS extension discovery, the handshake is exactly the one from DTLS.

3.1.2 Advertising interfaces

Like in MPRTP (see Section 2.2.3), we can explore 2 options for the addresses advertisement : in-band or out-of-band signalling. For MPDTLS, we made the choice to consider only the in-band communication for the following reasons :

- Lower overhead: we do not need any additional protocol or to set up another channel of communication.
- The addresses must be communicated securely. We already have a secure channel with DTLS, it is thus unnecessary to do another handshake with TCP/TLS.
- The reliability is not a strong requirement. The message carrying all the addresses may be lost, this is not vital for the communication. Moreover, with a small retransmission strategy, it will eventually reach the destination.

Once the handshake is finished and the initial subflow is established, the two hosts can advertise new interfaces available for other subflows. This is done within the `ChangeInterfaceMessage` (CIM), a packet carrying multiple addresses. This packet is carried as a DTLS fragment and thus is protected in the same way the Application Data are. The structure of the `CIM` packet is shown in the Listing 3.2. We use 16 bytes for the address to be IPv6 compliant. IPv4 addresses can be mapped to/from IPv6 format following `RFC4291`[33]. This mapping allows us to keep the size of the packet relatively small when you compare it with the one from MPRTP (Listing 2.1). Of course DNS names are not supported. It could be interesting to let the other host do the DNS resolution, but for now we think it is not needed. We can always extend the design if a need for other kind of interface advertisement is observed.

```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 epoch;
    uint48 sequence_number;
    uint16 length;
    select (ContentType) {
        case change_interface: ChangeInterfaceMessageFragment; // New field
    } fragment;
} DTLSPlaintext;

enum {
    change_interface(TBD, 42 in dev), (255)
```

```
} ContentType;

struct {
    byte reply;
    if (reply) {
        uint48 ack;
    }
    byte number_addresses;
    NewAddress addresses<1..2^8-1>;
} ChangeInterfaceMessageFragment;

struct {
    byte address[16];
    uint16 port;
} NewAddress;
```

Listing 3.2: Change Interface Message

The CIM contains all the addresses a host wants to share. We decided to always transmit all the addresses to provide redundancy. Also, to be sure we do not lose potential paths, the CIMs are retransmitted in case of loss and so are acknowledged. To avoid wasting resources, we use the acknowledgement to transmit the list of addresses of the receiving host. Doing this way, we are sure that each host knows the exact configuration of the other at any time (once the first host received the acknowledgement). This strategy has also been chosen for the retransmission in MPRTP [6].

An example of the CIM exchange is shown in the Figure 3.1. The message is structured as follows: the fact that it needs an acknowledgement or not (reply bit), the total number of interfaces we are advertising and the list of interfaces (following format presented in Listing 3.2). For the sake of clarity, the DTLS informations (such as the epoch, version,. . .) are not represented on Figure 3.1 and the sequence numbers are shown between chevrons.

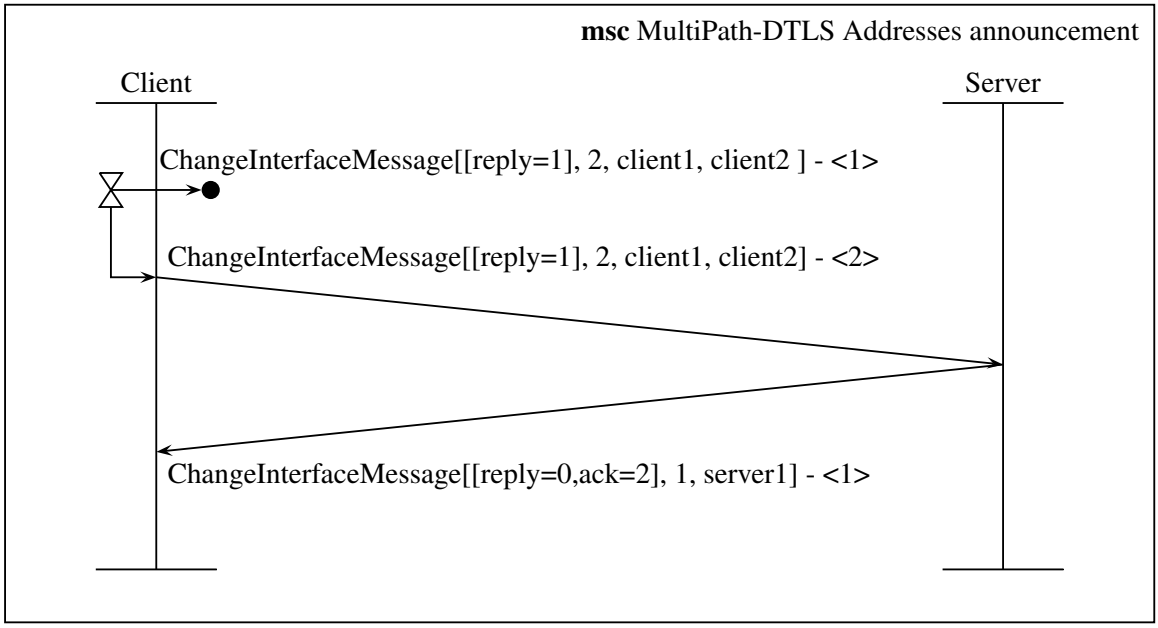

Figure 3.1: Example of Change Interface Message use

To avoid an infinite exchange of CIMs, a reply bit is needed and placed into the header of CIM to differentiate a new message from an acknowledgement to a previous one. Also, to know which CIM request is acknowledged, its DTLS sequence number is added in the response. The client must store the sequence number of the last CIM emitted. When a CIM is received but the host realizes the stored sequence number doesn't match the received ack, it must retransmit its own CIM.

3.1.3 Retransmission strategy

A CIM message must be retransmitted if the reply is not received because the packet may be lost. As we do not know directly if the reception is successful, we could use a timeout to retransmit the CIM after a certain period of time. But if the other host is definitely dead, we will send packets for nothing before we figure out. We could point out the fact that the knowledge of the interfaces is only needed when the host is really using them (i.e. sending packets).

Therefore, we could think of a mechanism which retransmits only when there is evidence that the other is still alive : when we receive application data packets. When the first CIM is sent, we set a flag to 1. This flag is put back to 0 as soon as we receive a CIM with `reply = 0`. But if we receive an application data with the flag still on, we process the data and retransmit the last CIM. An example is presented on Figure 3.2.

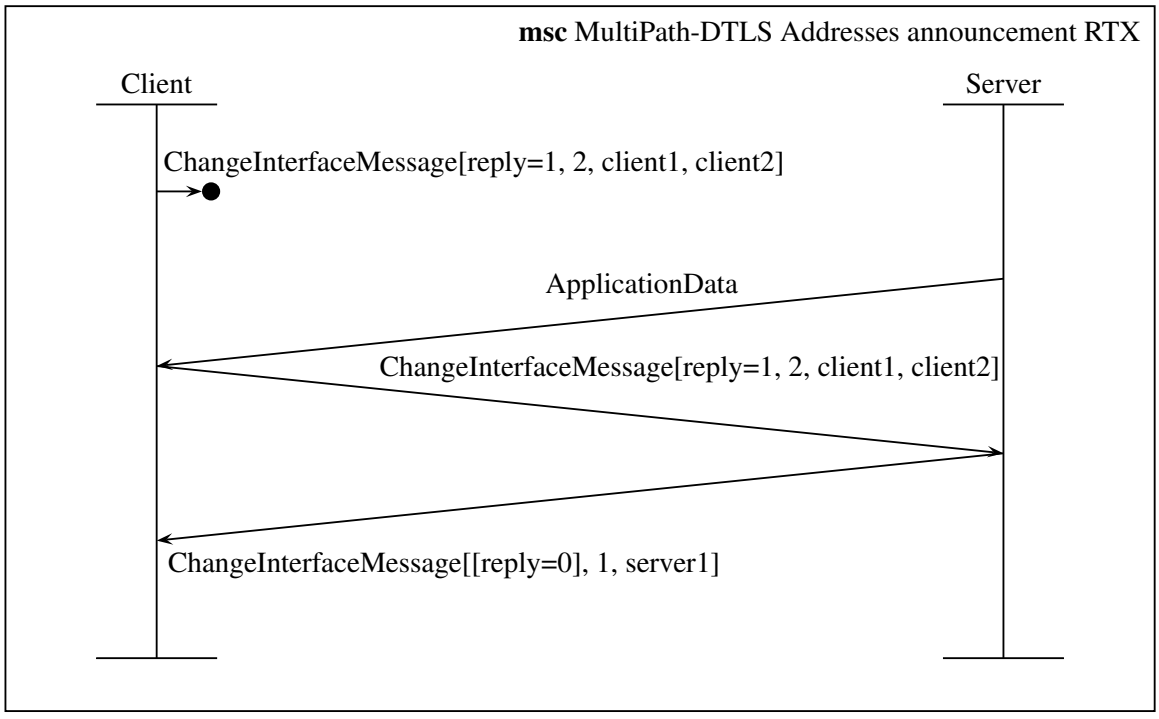

Figure 3.2: Example of possible Change Interface Message retransmission

Of course if some packets were already on the line before the server has received the CIM, we will retransmit many times the CIM and therefore waste bandwidth.

Our solution is somehow a mix between the pure timeout and the direct retransmission after every packet. When we first send the CIM, we store a timestamp in a variable. For every packets received, we check if

the elapsed time is enough and, in this case, we retransmit the packet and reset the timestamp to the current time. The "enough" is customizable, we could set a default value of 2 RTT for `CIM` retransmission. The main benefit of checking only when we receive packets is that we don't waste bandwidth for dead links and we don't need parallelization mechanism (e.g. interruptions or threads).

This strategy will be used for all retransmission of control packets with possibly different thresholds for the timers.

3.2 Secure subflows setup

In the first design of our protocol, we did not use any handshake to attach new subflows to the global connection. Instead, we were creating all the possible subflows by combining the host and remote addresses, directly initializing the sockets in connected mode. However, this solution is not suitable in various situations, like NAT-traversals or short-living interfaces.

Other multipath protocols don't need to explicitly start a new flow. That's the case for MPRTP, the interfaces are just advertised and all possible combinations of 2 interfaces are then acceptable. But MPRTP doesn't have any security concerns as we do.

Using not connected sockets could present a potential weak point for a DoS (Denial of Service) attack. If an attacker could guess IP addresses of the server, she can send garbage under the form of traditional DTLS packets. This could get computation costly since it forces the server to establish the packet authenticity and involves cryptographic operations. This is cheaper if we use connected sockets because packets coming from other addresses or ports than the connected ones are immediately dropped by the UDP stack. Also, this problem does not appear when the involved packets are not cryptographically signed, like the handshake packets. The solution presented in the Section 3.2.2 allows to overlook this issue as it is based on the shortened DTLS handshake.

To address this issue, we define three scenarios of subflow creation and the solutions to cope with.

3.2.1 Make-before-break sub-flows

We want to set up a new subflow when we have at least one other subflow alive. In this situation, we will use the secure communication already established to negotiate the opening of a new connection. Only when the negotiation is completed, a new socket will be created and connected on both side, avoiding any possible DoS attack on the new interface.

To carry this request, we need a new type of packet : `wantConnect`, whose structure is shown on Listing 3.3.

```
enum {
    want_connect(TBD), (255)
} ContentType;
```

```
struct {
    NewAddress addr_src;
    NewAddress addr_dst;
    byte opts;
} WantConnectFragment;

struct {
    byte address[16];
    uint16 port;
} NewAddress;
```

Listing 3.3: WantConnect message

The field `opts` can specify options for the incoming connection. The first bit of `opts` is telling if the host accepts or refuses the connection and will only be present in the reply. A second bit is used as the backup flag. When this flag is set to 1, it means we want to send packets on this interface only if it is the only possible choice to keep the connection running. Typically, this would be the case for a 4G/LTE interface on a phone because it costs much more than the Wi-Fi. The 6 remaining bits are unused for now but may be useful in the future.

The host who received a `wantConnect` must acknowledge the reception using a `wantConnect Ack` packet (see Listing 3.4). Messages can be lost and we need to know to which packet the acknowledgement is referring to, so we include the sequence number of the corresponding packet (i.e. DTLS sequence number). The options are also included to give the opportunity to the other host to accept or deny some of them. In particular, the first bit is set to 1 if the host refuses the connection and zero otherwise.

```
enum {
    want_connect_ack(TBD), (255)
} ContentType;

struct {
    uint48 ack_seq;
    byte opts;
} WantConnectAckFragment;
```

Listing 3.4: wantConnectAck message

Figure 3.3 presents how it will take place when a server has 2 interfaces and the client only one. First the traditional DTLS handshake is established between the client and the public interface of the server (S1). By a `CIM` exchange, the client is aware of the existence of a second interface (S2). It then sends a `wantConnect` request and if this message is correctly received, the server will set up its second interface to receive packets from the client. After this exchange, heartbeat messages will take place to assess the availability of the link. The frequency must be defined according to experimentation. At this point, we recommend to send a heartbeat message at least every 5s. The implementation must use a back-off strategy to prevent waisting resources on a dead link. If for some reasons, the interface is not reachable from the client, then the server will find out it never receives any heartbeat response and will close the socket. Otherwise, a new subflow has been set up and both hosts can use it to communicate securely.

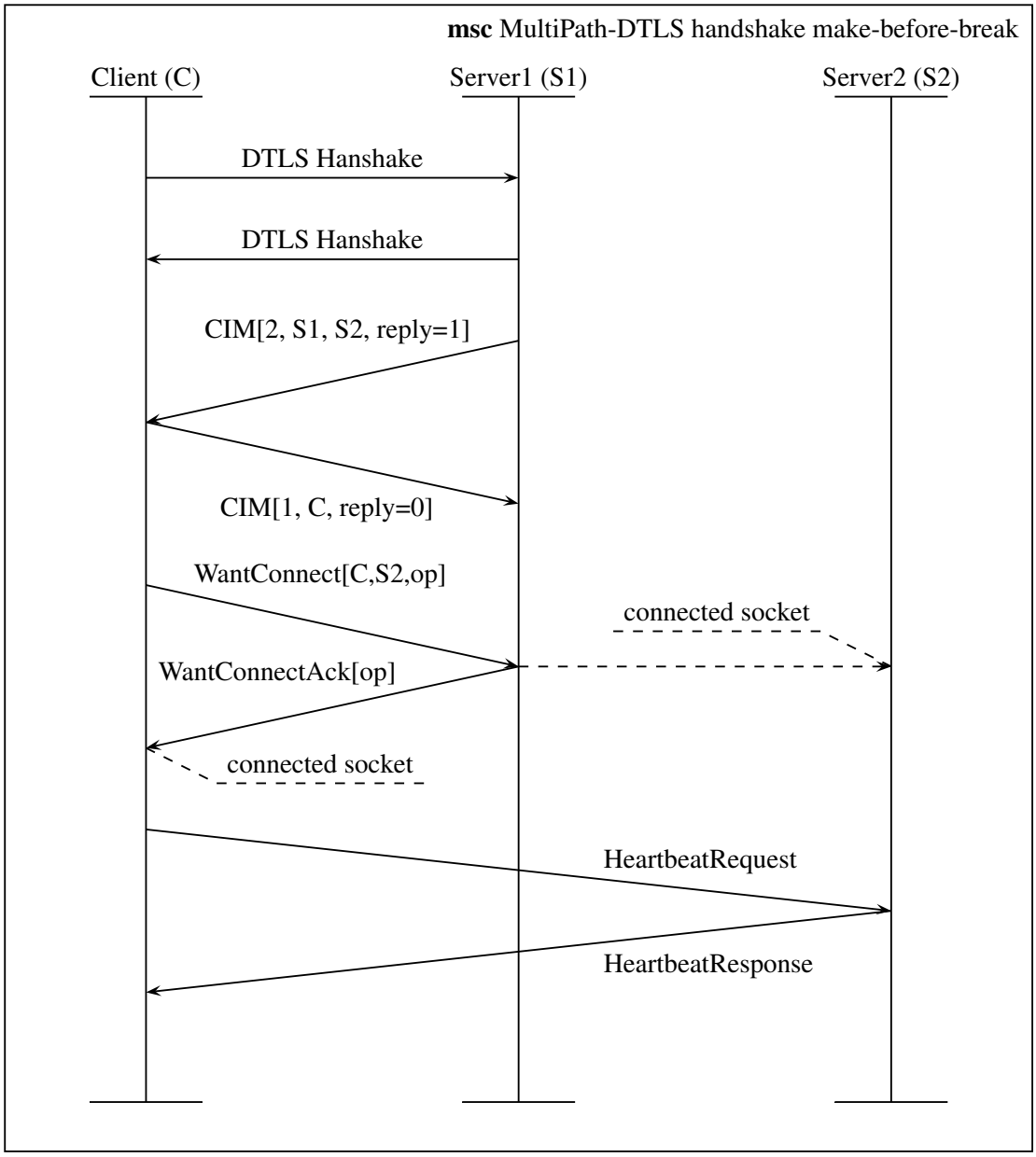

Figure 3.3: Example of new sub-flow establishment when another connection is alive

When another connection is available, this is the best method to establish a new sub-flow since it is way faster than a complete handshake. Note that the two new messages introduced are secured as any other DTLS message and therefore cannot be forged or replayed.

3.2.2 Break-before-make sub-flows

Unfortunately, the solution presented in the previous section is not applicable in every situation; for instance, when a smartphone is connected with its Wi-Fi interface and the access point becomes out of range for any reason, the smartphone will then toggle the 4G interface. It cannot use the procedure explained in the previous section as it does not have an active subflow anymore. To solve this problem, the smartphone can simply use the Session Resumption mechanism available in (D)TLS, resuming on

both sides the state of the session as before the link failure. It is also important that the smartphone warns the server that the Wi-Fi interface is no more available by sending a `CIM` as soon as it re-establishes the connection. This use case is presented on Figure 3.4.

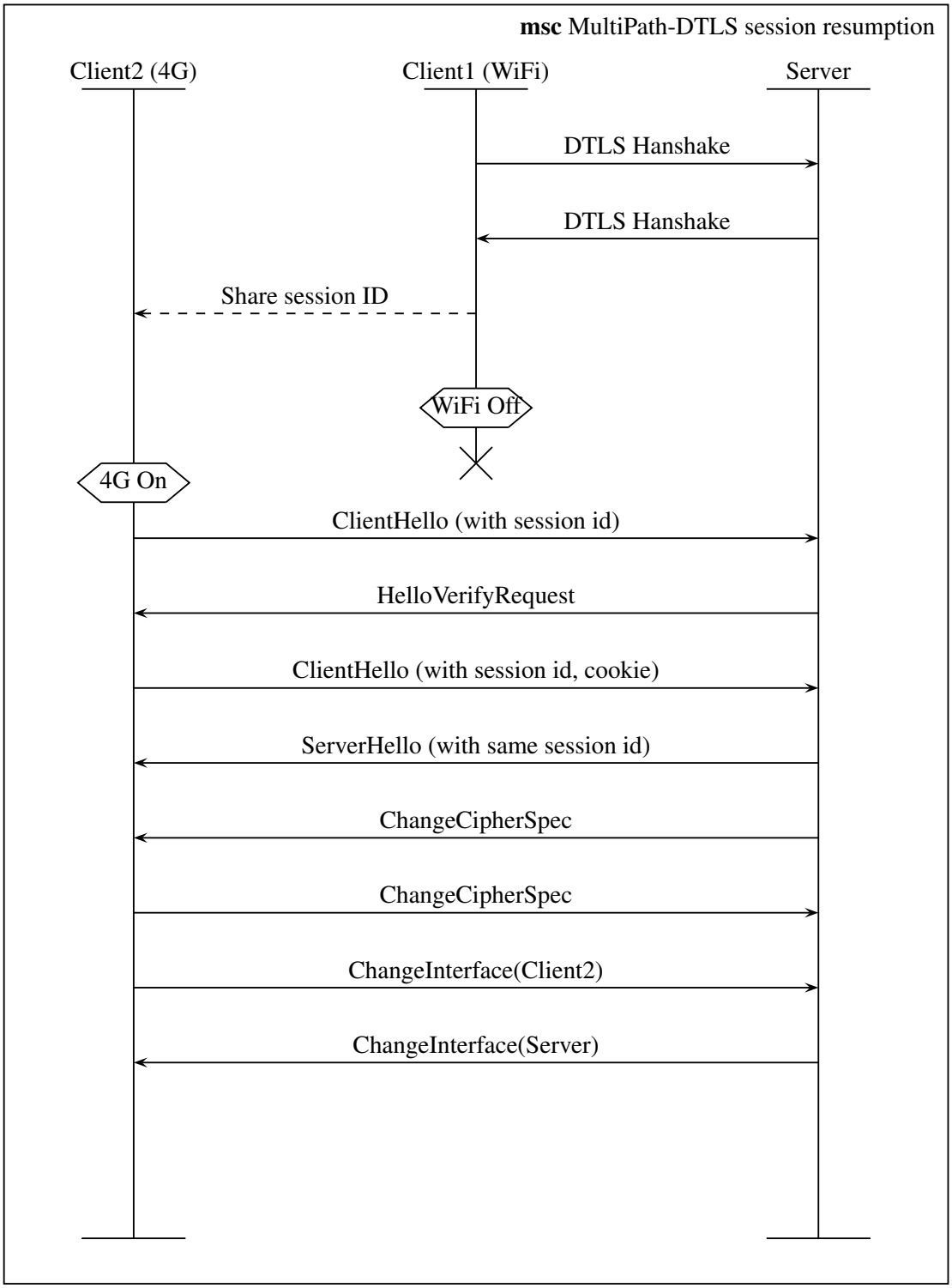

Figure 3.4: Example of how session resumption takes place

Note this is only possible because Client1 and Client2 are the same application and therefore share the session ID given by the server. The handshake is faster if the session is still on the server's cache because

no certificate exchange is needed and the master key is the same as before.

This solution presents the same security properties than the standard Session Resumption proposed in the RFC5246[1], Section 7.4.1.2. The RFC doesn't explicitly limit the session resumption to come from the same IP/port as before but it appears that some implementation do it. It was the case for Firefox and was criticized as a bug by N. Modadugu who is the co-author of DTLS[1]. It can't be considered as a weakness since even if an attacker can steal the session ID, she doesn't know the master key and therefore won't be able to generate a correct ChangeCipherSpec.

If the server clears the session cache in the meantime, the full DTLS handshake will take place and the server and the session will start without any other known remote interface. This is another reason to impose a CIM emission as soon as the subflow is up.

Moreover, to make this solution working, a thread or a process must be constantly listening the interface even if a connection is already initiated. This was not a requirement for the *WantConnect* solution.

Finally, we can say that this last solution is more generic than the previous one, as it can be used in every situation, when the WantConnect can only be used if a sub-flow is already established. Nevertheless, this method could potentially complicate the implementation if a flow is up and you try to do session resuming with another flow. You will have two separated DTLS sessions and the expected behavior is to merge these two into only one session. It would imply to use some communication channel between processes (shared memory for instance) and to merge objects together. Our experience working with an SSL/TLS library shows it won't be easy. So, the best solution to establish new flows when at least one other flow is alive stays the WantConnect method.

3.2.3 NAT-traversal subflows

However, an overview of the possible scenarios would not be complete without taking into account the NAT that are widely deployed in home networks. To do so, we can reevaluate the solutions proposed in the previous sections to see if they can fit in presence of NAT.

Make-before-break

Unfortunately, the initialization of new subflows as described in 3.2.1 is not possible if at least one host is behind a NAT. Indeed to create a connected socket, one host must know both the IP address and the port number of the other host. While the IP address may be obtained by some external online tool, the port is attributed by the NAT when the flow is created. So, the NATed host has no clue to guess that port number.

To perform efficiently NAT traversal, we must rely on an external protocol for session establishment. Such a tool is available under the name of Session Traversal Utilities for NAT(STUN) and is described in

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=415196

RFC 5389[34]. It uses an external server with a public IP to play the role of intermediary and get public IP and port number for a particular session.

Other protocols actually delegate this task even if no NAT is present. This is the case for RTP and thus MPRTP, the IP addresses and ports are obtained typically via Session Initiation Protocol (SIP) [35].

Therefore, we let the application obtain the public IP address and the port of every interface via another protocol. In the prototype we have developed to test our implementation, this feature has not been implemented since the objective was not to produce a commercial tool to use at home.

Break-before-make

The session resumption method is not altered by the NAT. The only constraint is that the handshake must be initiated by the NATed host, to allow the NAT-holing and the correct address discovery. This is true as long as there only one host behind a NAT or one of the NAT is correctly configured forward the traffic towards the host. In presence of two misconfigured NATs, the same comments of Section 3.2.3 apply.

3.3 Feedback on subflow

Last but not least, each packet must be sent on a single flow. Thus, one needs to dispatch the packets over the subflows, and this is the role of the scheduler. But, to be able to do its work efficiently, the scheduler must be aware of the subflows health. To implement this new feature, we propose to add to DTLS a feedback mechanism, gathering various information such as the forward delay, the drop rate of the link or the global reorder rate. Crossing the information we receive from different subflows will allow the scheduler to dispatch packets efficiently.

3.3.1 Forward delay estimation

The transmission delay is an important measure when we want to balance a load over multiple subflows. But, in our first design attempt, we used the RTT to measure this delay, estimating the one-way delay to the half of the RTT. However, as a majority of the links over the Internet are asymmetric, it sometimes leads to major differences between the real one-way delay and our estimation. We thus needed to reconsider the way to compute the delay.

The most intuitive way to compute the time taken to realize a task is to compare the time before its beginning and after its completion, which makes no sense when it comes over the network. As the task starts on one host and finishes on the other, we are subject to the clock synchronisation.

However, Fei Song et Al. propose in [36] a practical solution that suits our needs. The idea is simple: we don't need to know the exact One-way delay of a subflow, we just need to be able to compare the delays between the different subflows. So, we can compute the transmission delay of a subflow as the difference between the sending time and the receiving time, with a ΔT term being the clock desynchronization

between the two hosts. This ΔT is only the clock difference that could exist between the two actors of the connection and it does not hide any delay or jitter caused by the transmission itself. As the two end-points of all the subflows are the same, this ΔT is assumed to be constant over all the subflows.

Once we know how to estimate the forward delay of each subflow, or at least how to rank them on this criteria, we can easily create a mechanism to compute this estimation, it is illustrated in the Figure 3.5. To avoid overhead on each DTLS Application Data packets, we took the decision to use new dedicated packets to compute the forward delay. Each host will periodically send heartbeat packet containing the current timestamp. When the other host receives it, it can compute the transmission delay modulo ΔT. For the sake of simplicity, only the average of the delay computed will be transmitted in the `feedback` report, as presented in Section 3.3.3. The ΔT is considered constant over time because the clocks are increasing in the same way on all the hosts. As ΔT is constant, it does not introduce deviation in the calculation of the transmission delay average.

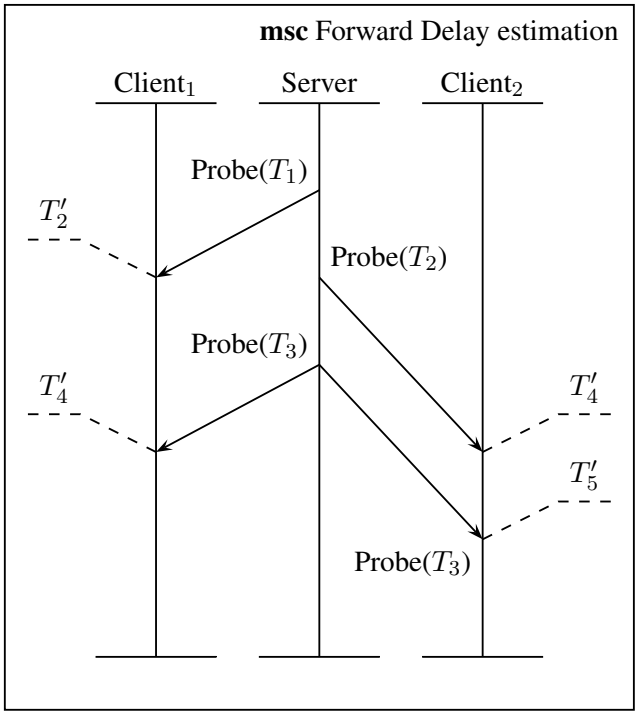

Figure 3.5: Forward Delay estimation mechanism

To compute the forward delay, we propose to proceed as follow:
The host (*Server* in the Figure 3.5) sends periodically `Heartbeat` packets containing its current timestamp (T_x).
Once received by the other host (Client$_{\{1,2\}}$), the latter can compute the transmission delay of this particular packet,

$$FD_1 = T_1 - T'_2 + \Delta T$$
$$FD_2 = T_3 - T'_4 + \Delta T$$

and update its estimation using EWMA[a]:

$$EFD_i = EFD_{i-1} * \alpha + FD_i * (1-\alpha) [+\Delta T]$$

with $\alpha = 0,875$ (Jacobson's algorithm).

[a] **E**xponentially **W**eighted **M**oving **A**verage

We preferred to use the EWMA mechanism that is already use for the RTT estimation into TCP. In this way, the new value can influence enough in case of sudden change, but cannot completely mess up the average in case of isolated measurement outlier.

Finally, this is this EFD_i value that is transmitted to the emitter through the feedback message, as we can see in the Listing 3.5. The sender thus knows the forward delay estimation of each subflows and its scheduler can take better balancing decisions.

3.3.2 Loss rate

Unlike in TCP, we don't receive ack for every packet correctly transmitted. Moreover because DTLS is based on UDP, a loss is actually a normal event. To compute the loss rate, we must then add a new mechanism to support feedback. This is done by regularly sending `feedback` packets from the receiver to the sender. More details about this mechanism including packet structure are presented later in section 3.3.3. When we talk about sender and receiver, we divide the DTLS connection in two one-way half-connections. So if we put things back together, each host will play the role of a sender and a receiver.

In this feedback, we do not want to acknowledge every packet received. Instead we give some information about what we received in the time frame. It includes :

- the number of packet received
- the minimum and maximum sequence number received

As receiver, by transmitting these information back to the sender, we give the sender the ability to estimate the loss rate for this particular subflow. The minimum and maximum sequence numbers received alone are not enough to assess very accurately the loss rate but this is a way to keep the packet size constant. Moreover, if we consider the reordering on a single path as a rare event, we can obtain the loss rate by

$$LR = \frac{packets_{sent} - packets_{received}}{packets_{sent}}$$

where $packets_{sent}$ is maintained by the sender and $packets_{received}$ is extracted from the feedback. This loss rate will be added to the global loss rate of the flow using the same EWMA mechanism as for the forward delay. This will reduce the impact of losses on a very short period of time.

Note that the sender must only keep track of packets with a sequence number greater than the last max sequence number received from the last feedback. In this way, the space used to store these sequence numbers will be kept reasonably small. In the case we don't receive feedback anymore, we will progressively send less and less packets to this address until we stop sending. Therefore, it is not possible we exceed sender's memory simply because the receiver is dead. To do so, a back-off timer will be set up for the heartbeat messages. Such a timer will increase exponentially and above a certain threshold, we will consider the link as broken. Nevertheless, a good scheduler will stop sending packets before the threshold is reached by looking at the number of packets not yet acknowledged.

If one interface goes offline and at least one other link is still available, a CIM must be send to warn the other host. The latest draft of MPRTP [6] Section 7.4 relies also on the communication between the host and the endpoint to explicitly discard one subflow.

3.3.3 Feedback reporting

Figure 3.6 presents an example where feedback takes place once the communication is well established. After a reasonable number of packets is received (2 in the example), we trigger the emission of a `Feedback`.

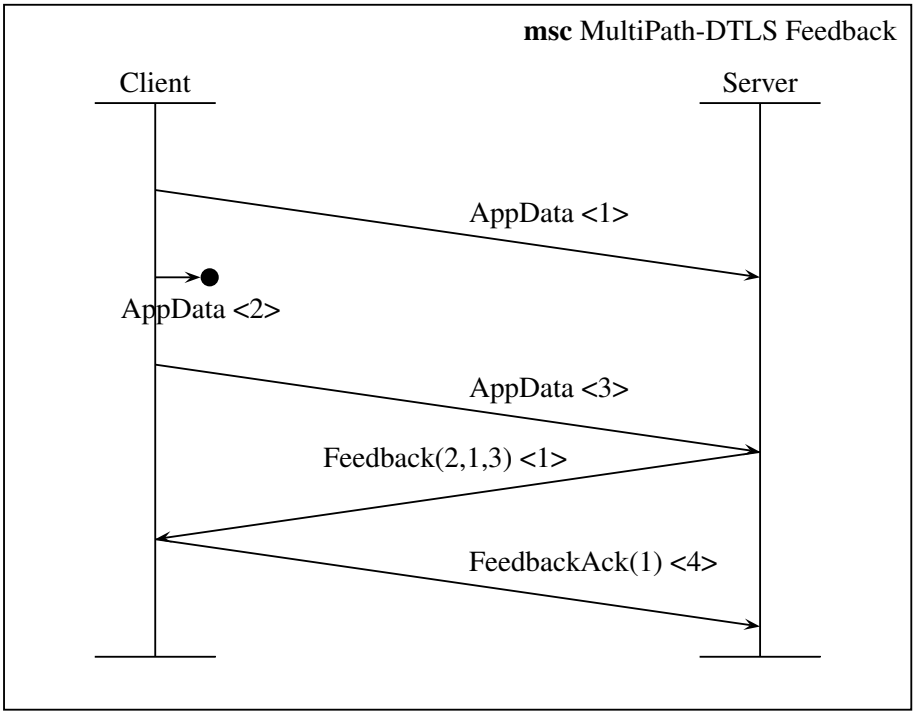

Figure 3.6: Feedback flow

The structure of `feedbackMessage` is presented in Listing 3.5. In the example, `Feedback(2,1,3)` means that we have received 2 packets since the last acknowledged feedback. The minimum and maximum sequence numbers received are 1 and 3 respectively. The client replies with a `feedbackAck Message` carrying the sequence number of the corresponding `feedbackMessage`.

The size of the sequence number is directly taken from the `RFC6347`[8] while we consider 8 bytes enough to count the packets. The threshold which triggers the transmission of a `feedback` must be fixed way before this limit to provide useful information even at Gigabit speed. But in case some feedback or `feedbackAck` is lost, we must handle more than usual values. Indeed, as long as no acknowledgement has been received, the receiver will continue to send incremental feedback (i.e. the new feedback contains information about packets already reported in old feedback but not acknowledged).

```
enum {
    feedback(TBD), feedback_ack(TBD), (255)
} ContentType;

struct {
  uint64 received_packets_count;
  uint48 min_sequence_number;
  uint48 max_sequence_number;
  uint64 average_forward_delay;
```

```
} FeedbackFragment;

struct {
  uint32 feedback_sequence_number;
} FeedbackAckFragment;
```

Listing 3.5: Feedback and Feedback Ack messages

A `feedbackMessage` is actually a modified DTLS `ApplicationData` packet. In particular, it contains a signed sequence number. The latter can be used in the `feedbackAckMessage` to identify uniquely this packet. In case of retransmission/loss when we receive a feedbackAck, we always know to which feedback it refers to.

Feedback reporting is done on a regular basis. The threshold to send a feedback is to be defined accordingly to the link bandwidth. On a gigabit speed link for example, we will not send feedback every 10 packets but this could be the case in a low bandwidth, high latency environment.

3.4 Impact of TLS 1.3 modifications

If we look at the latest version of the draft on TLS 1.3 [13][2], the section 1.2 reports all the major differences from TLS 1.2 which is the base for DTLS 1.2.

Many modifications have an impact on the handshake phase. This is the case for the choice to remove the support for static RSA and DH key exchange. It will certainly improve the overall security of the communications but it does not interfere with our design. Indeed, we modify the handshake only to advertise the presence of our extension. This will remain valid with the current draft since a spot is still available in the `ClientHello` to carry extensions (see Section 7.3.1.1 of [13]). All other TLS extensions are using the same mechanism so we have good reasons to think that the next version of TLS will keep supporting it.

One of the biggest change is probably the end of the renegotiation. This feature has brought some important weaknesses (see Section 1.5) and therefore will not be part of TLS 1.3. Fortunately, we do not use this feature but only the session resumption for the break-before-make scenario (see Section 3.2.2). There is a big difference between these two concepts. While session renegotiation redefines all the parameters for the session, session resumption reuses the same cipher suites. The keys used to encrypt and authenticate the packets must be the same as before the resumption. As a consequence, session resumption is considered as secure and will be supported by TLS 1.3.

As conclusion, all the changes made in TLS 1.3 up to now have no effect on our current design. This is mainly due to the fact that we don't modify existing packets but create new ones. Moreover our presence in the handshake is limited in practice. We may expect a version 1.3 of DTLS will follow the TLS one and we have no reason to think that our design for MPDTLS will not be compatible with this future version.

[2] Version 5 as we write these lines.

4 | Implementation

This chapter focus on the implementation of our extension: Multipath DTLS[37], whose design was presented in Chapter 3. We describe first the library we chose to modify, detailing the internal workflows such as the handshake or the packet emission and reception by following a use case (from the tunneling application we developed for our tests, presented in Section 5.1). Then we explain how we handled some programming aspects such as the thread-safety and the retransmission timeouts. Finally we detail the modifications made to add the multipath ability and the statistics gathering, as well as presenting the internal structures supporting these features. We also show the various issues we faced during the implementation and how we solved them.

4.1 Choice of library

Since we designed MPDTLS as an extension for DTLS, it was logical not to implement it from scratch but to start from an existing implementation of DTLS. Several libraries implement the latest version [38], but we chose wolfSSL [39] (previously CyaSSL) as our starting point.

The choice was driven by the following criteria :

- the code readability
- the documentation
- existing examples
- the library size

Unlike OpenSSL, wolfSSL contains a small number of files to handle SSL/TLS and DTLS. The objective of this library is to be as light as possible to allow its integration in embedded systems. As stated on its official website [40], the library is up to 20 times smaller than OpenSSL [7]. Documentation and working examples are provided to help its use.

For information purpose, a summary of all the modifications made inside wolfSSL is presented in Appendix D.1. In addition, we have modified the DTLS dissector of Wireshark[41] to recognize our new packets and debug more easily (see Appendix D.2).

4.2 Library calls

In this section, we show how to use wolfSSL, how to set up the library and create a DTLS session and how to transform an existing DTLS code to use MPDTLS instead. Then we dive deep into the library code to see how the handshake is done internally and how the packets are sent and received.

It is important to remind wolfSSL is a library and thus does not have any existence outside the calls made and the structure stored in variables in the client code.

4.2.1 wolfSSL Context initialization

The code to initialize the library context is similar for the client and server side.

First, the context structure must be initialized. WolfSSL provides a function to do so, `wolfSSL_CTX_new`, that takes a `WOLFSSL_METHOD` as argument. This method represents the protocol that will be used in the underlaying sessions. In our application, we obviously use DTLS as protocol.

```
wolfSSL_Init();

WOLFSSL_CTX *ctx;
WOLFSSL_METHOD* method = wolfDTLSv1_2_client_method();
if ( (ctx = wolfSSL_CTX_new(method)) == NULL){
    // error handling
}
```

Once the `ctx` *object* is created, we can call various functions to set up some parameters, like:

- the addresses advertised by the host,
- the cipher suites that the host can use to exchange the security parameters during the handshake and to protect the conversation later,
- the certificate and corresponding private key to authenticate the host (this is optional for the client),
- the Certification Authority certificate that allows the host to check the validity and integrity of the received certificates.

Other parameters can be set and they are all well explained in the wolfSSL documentation[42].

4.2.2 wolfSSL Session creation

Client side

Once again, wolfSSL provides a simple function to create sessions, `wolfSSL_new`, which takes a context in argument. In this way, the session is initiated with the parameters set up in the context previously

created. To turn on Multipath DTLS, we have added a simple function, `wolfSSL_UseMultiPathDTLS`, to enable or disable it.

The next step is first to provide the library with a freshly created socket. It does not have to be connected at this point. Secondly, we need to set the server address through the `wolfSSL_dtls_set_peer` call.

```
wolfSSL_set_fd(ssl, *sockfd);

if(wolfSSL_dtls_set_peer(ssl, serv_addr, sz)!=SSL_SUCCESS){
    //error handling
}
```

In case of session resumption, the session can be restored with the `wolfSSL_set_session`. The session will be automatically resumed if it is still present in the server session cache. If the session has been wiped out from the cache, the library will ignore this call. The `WOLFSSL_SESSION` object can be retrieved from an established DTLS session by using the `wolfSSL_get_session` call.

```
if(sess != NULL) {
    if(wolfSSL_set_session(ssl, sess)!=SSL_SUCCESS) {
        //error handling
    }
}
```

Finally, we can connect the DTLS client to the server. If the `wolfSSL_connect` call is successful, we can check the correct set up of the MPDTLS extension with the `wolfSSL_mpdtls` function.

```
if (wolfSSL_connect(ssl) != SSL_SUCCESS) {
    //error handling
}
```

Server side

From the server point of view, the session creation is almost the same. The first difference is that the server must listen an unconnected socket (like every server) and create the session only once a message have been received on this socket.

The other difference is that the server does not have to call `wolfSSL_dtls_set_peer`, but it has to use `wolfSSL_set_fd` with a connected socket (from its own address and port towards the sender of the triggering packet). Finally, the server calls `wolfSSL_accept` instead of `wolfSSL_connect` to start the handshake.

4.2.3 Handshake

The handshake that is executed on `wolfSSL_accept` and `wolfSSL_connect` is the standard handshake defined in the RFC6347[8] and illustrated on Figure 1.3. As the handshake consists of several

messages sent back and forth between the client and the server, the work-flow behind this handshake heavily relies on the packet reception and emission processes. These processes are detailed in the Sections 4.2.4 and 4.2.5 respectively.

Internally, the `wolfSSL_accept` and `wolfSSL_connect` functions are built in the same way, the sole difference being the expected and sent packets. A variable is initialized at the beginning of each function, and modified on reception of each message. This variable contains the state of the handshake, and so determines the messages that should be read or sent. If an unexpected one is received at any point, two behaviors can occur, depending of its nature. Either it is kept in a buffer for later use, if it presents evidence of reordering, or the handshake is aborted and the function returns with the corresponding error code. This error code informs the application about what gone wrong. Otherwise, the function returns only once the handshake is successfully completed.

4.2.4 Packet reception

As soon as the `WOLFSSL` structure is correctly instantiated and the handshake has been successfully completed, we can start listening to the connection. This can be done by calling the `wolfSSL_read( WOLFSSL*, void*, int)` function, whose internal flow inside the library is detailed in the Figure 4.1. Note that the mutex calls are not part of the original wolfSSL code, and are explained in Section 4.3.

When the function `wolfSSL_read` is called, the process enters the library, dives into the `Receive Data` and `ProcessReply` functions and will eventually make a blocking call into the `CBIORecv` macro[1] to listen the socket. When a new packet arrives, the process is woken up and it copies the packet into an internal buffer. Once the copy is made and the pointers are correctly set, the process can treat the packet.

First, it reads the `RecordLayer` header, containing the length of the packet and the record type. The latter is one of the `ContentType` values defined in the RFC[1]. During this verification, it also defines, based on the `ContentType` value, if the packet is encrypted and therefore needs to be deciphered before being handled.

Then, after the decryption of the packet, it is dispatched to the handling function corresponding to the `ContentType`. These functions are called following the pattern "`Do`*TypeName*" (e.g. *DoApplicationData* to handle application data packets). For the sake of clarity, these two steps (decryption and all possible `Do`functions) are not represented in Figure 4.1.

In the original implementation of wolfSSL, the only Record type that could be encountered at this point was `Application Data`. The function *DoApplicationData* applies some handlings on the packet like verifying the MAC or decompressing the packet if needed and finally copies the clear content of the packet into a buffer called `clearOutputBuffer`. Then, if the input buffer has been completely consumed, the process comes back to the function `ReceiveData`. This function copies the content of `clearOutputBuffer` to the application buffer passed in argument of `wolfSSL_read`, thus giving

[1] The low-level mechanisms to read and write on sockets are indeed encapsulated into macros selected following the compilation or configuration flags. This provides to the library some cross-platform aspects.

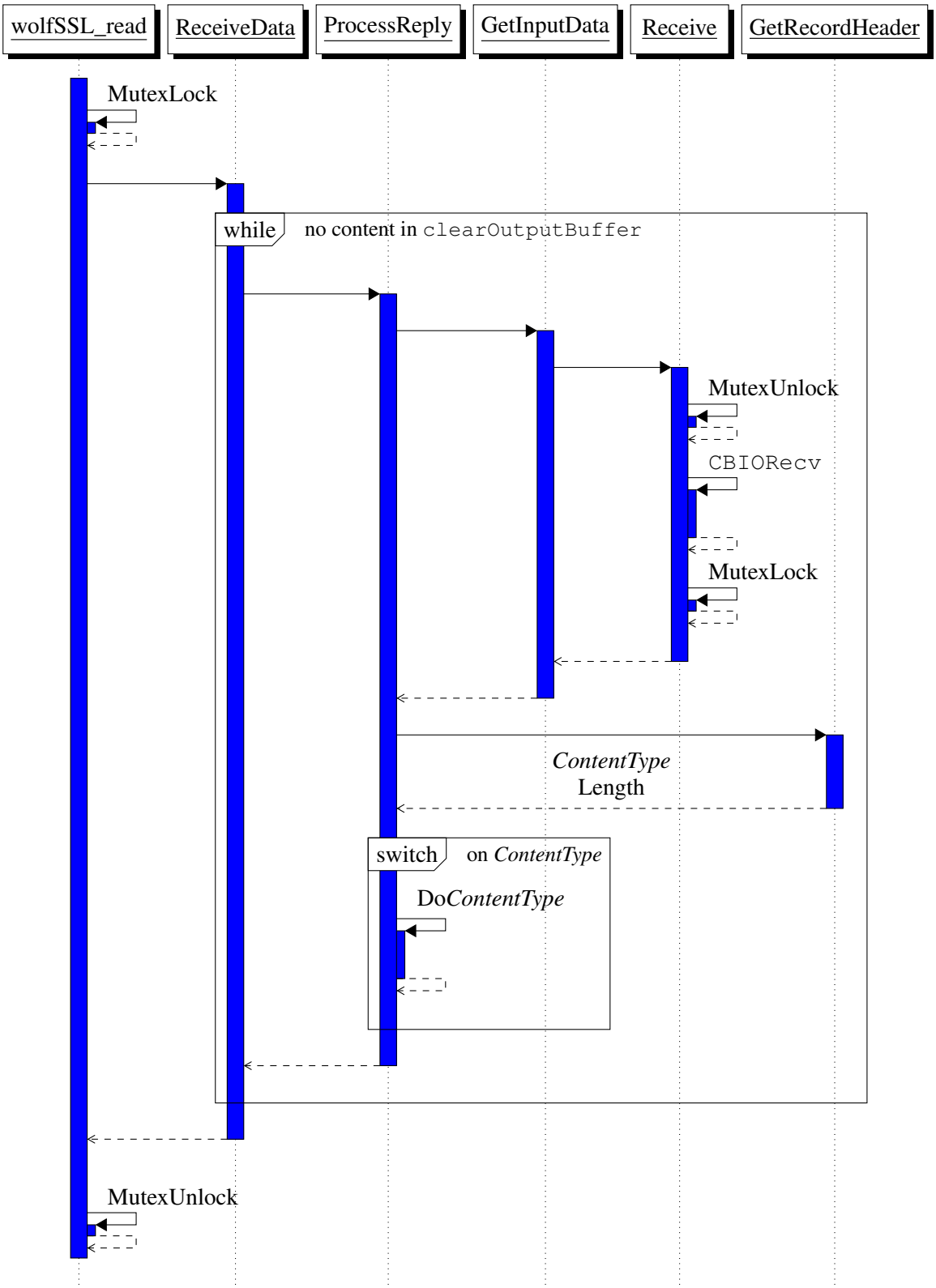

Figure 4.1: Major functions called on `wolfSSL_read()`

to the client application the content of the received packet. The library then exits and gives the control back to the calling code.

Adding new types

Adding the recognition of a new type needs two steps, after the definition of all the declarations needed in the header file: we start by adding the type in the switch of the `GetRecordHeader` function. We can also add indication about the packet type, such as whether the content is ciphered or not. Then, we need to add a new case for the type in the switch statement of the `ProcessReply` function, in which we make the call to a new, dedicated function `Do`*`TypeName`*.

Now that the library recognizes the new type and calls the appropriate `Do`function, we can implement the latter with the intended behavior. In our case, we use the `DoApplicationData` as a base for the implementation of all the `Do`function of protocol-level packet types (e.g. `Feedback`, `WantConnect...`). It is the easiest solution to get ciphering and authentication of the packets. But it is important to prevent these packets from leaving the library and being delivered to the application. To achieve this, we move the pointer marking the end of the `clearOutputBuffer` before the beginning of the current packet, erasing in some way the packet we just have treated.

4.2.5 Packet emission

The packets emission is even simpler, as we do not have to wait for some event, we trigger it. So, in wolfSSL, an application can send packets using the `wolfSSL_write(WOLFSSL*, void*, int)` function. As shown in the Figure 4.2, the process then enters in `SendData`, which is responsible of all the authentication and encryption stuff, in the original code of the library. We moved the code handling the MAC and encryption in a new function `SendPacket`, allowing us to use the encryption not only for the Application Data packets but also for our protocol-level packets. As for the receiving functions, we created a set of `Send`*`TypeName`* functions, in charge of building the corresponding packets before sending them. All these methods end with calling `SendPacket` which will then call `SendBuffered`. `SendBuffered` is the last function in the library before leaving the control to the concrete sending macro `CBIOSend`, and this is the place we choose to put our scheduling mechanism. The socket to use is chosen at the beginning of `SendBuffered`, just before the call to the sending macro.

4.3 Thread-safety

A major drawback of living at the application-level for a library is to be dependent on the application. For instance, all protocol-level packets need responses to ensure the reception since DTLS is not reliable. But, to trigger the internal mechanisms explained in the previous sections, the library must be in constantly a listening state. If an application needs to send some packets, which seems to be a normal use case of the library, it will need to use threads to read and write simultaneously. For this reason, we need to be sure that using the library in such a way will not cause troubles, especially since the library uses a

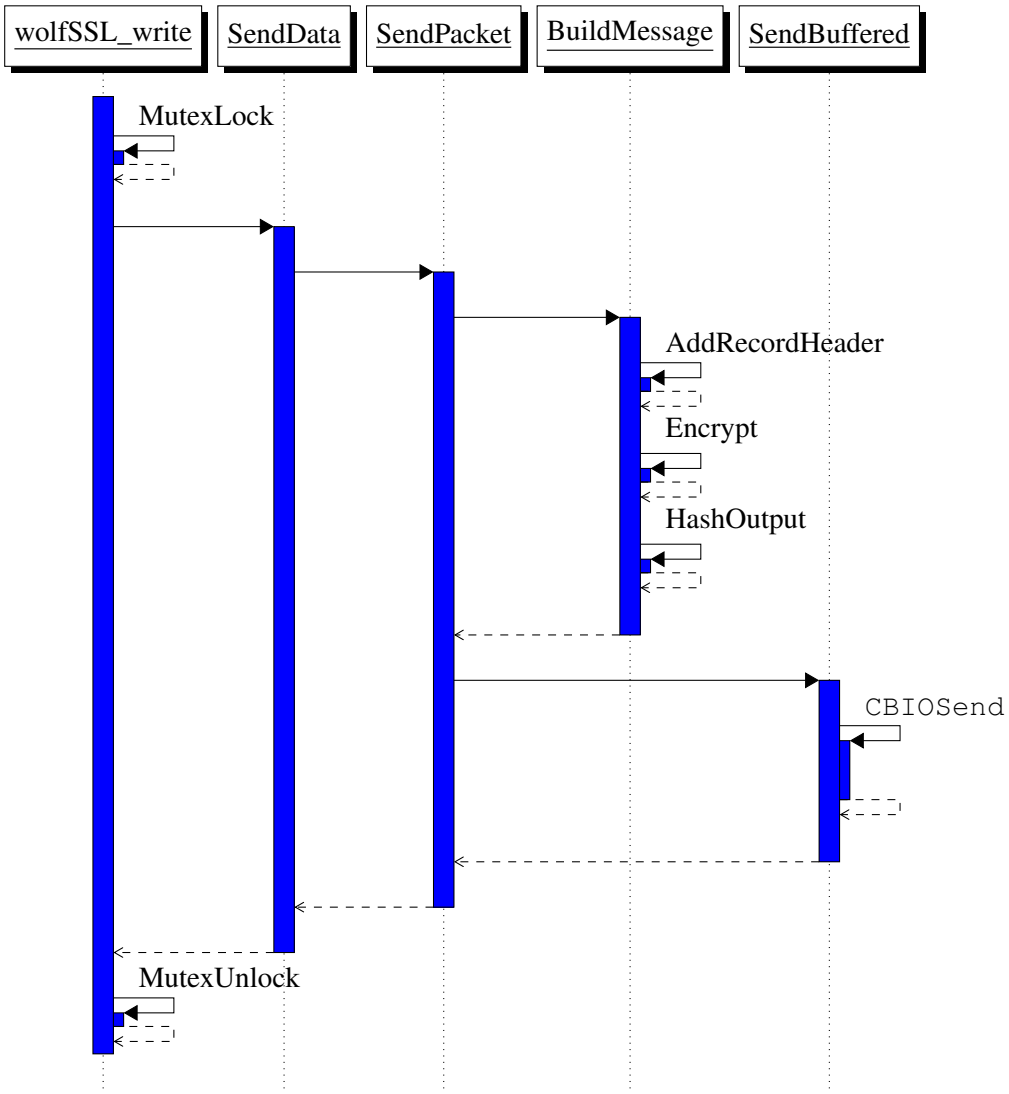

Figure 4.2: Major functions called on `wolfSSL_write()`

lot of internal buffers when it reads or writes. Because our handlings of protocol-level packets involve packets emission during packets reading, we considered the whole library code as a critical section, with a notable exception for the blocking call in the `Receive` function[2].

Thus, as with every critical section to be protected, we added a mutex to lock when entering either `wolfSSL_read` or `wolfSSL_write` and to unlock just before leaving these functions. This mutex is also unlocked just before the blocking call to `select()` and re-locked right after this function returns.

4.3.1 Note about processes

Using processes instead of threads is another solution, but processes have either to use different session credentials or to share memory to use the same session object. It is also possible to simply duplicate the session either through the session resuming or via copy-on-write session mechanisms, but just having

[2] As a call to `CBIORecv` is blocking, it prevents the application to write packets when the other thread is waiting for incoming packets, which would add an extra undesirable overhead.

the same session credentials is not sufficient for the library to work correctly. As we use statistics to choose the best path to send new packets, the session object must be completely synchronized between the processes, with all the risks of memory corruption due to the simultaneous accesses. We finally end up in the same situation as with the threads.

For the sake of simplicity, we recommend to simply use threads, as we ensure the thread-safety of the library.

4.4 Managing timed (re)transmissions

WolfSSL does not use interruption in its original version and we did not want to add this kind of control mechanism. This is a problem as we have to handle timeouts and retransmissions for the control-level packets. Note that by timeout we mean either that the packet has not been acknowledged by the other host or that the packet must be sent periodically and the last sending occurred too long ago.

We thus had to use other ways to detect that a packet needs retransmission. By checking actively after each packet sent, we can do without the interruption mechanism. This check is implemented with the addition of the call to the `CheckTimeouts` function, just before the return of `SendBuffered`. This function verifies that neither `Heartbeat` message nor `CIM` need (re)transmission. To know which packet needs to be retransmitted because it is in timeout state, we use two possible criteria. First, we base the timeout on the time elapsed since the last packet was sent. Second, we use as trigger the number of packets sent since the last packet of the considered type was sent.

4.4.1 Timeouts

With the timeout strategy, we are guaranteed to retransmit the packet within a delimited time window, provided that other packets are sent as we check the timeouts only at this moment. The implementation of this part is quite easy, only asking to store the current time when we send the packet. The verification then consists in simply comparing the difference between the stored time value and the current time with a defined timeout offset.

The protocol does not give any guarantee about the delivery of `ApplicationData` packet but requires that `Heartbeat` and `CIM` are transmitted reliably, so we use this strategy to ensure the retransmission. The `Heartbeat` messages are sent periodically and this mechanism can also meet this requirement.

Concerning our implementation, the timeval for the `CIM` is stored in the general structure of wolfSSL, while each subflow stores the timeval for its `Heartbeat`, since each subflow manages its own `Heartbeat` exchange.

4.4.2 Packets count

If the previous strategy guarantees some time limits on the trigger, the "packet-count" strategy allows us to have a more adaptive transmission regarding the actual traffic. The feedback primary purpose is directly related to the volume of traffic on a subflow and we decided to use this strategy for the `Feedback` packets, both for the transmission and retransmission in case of missing acknowledgement.

Altough the design does not require a reliable exchange of `Feedback` reports (see Section 3.3.3), we chose to reduce the number of packet to aggregate before sending feedback in case of incremental feedback. That is to say when the `Feedback` or `FeedbackAck` is lost. The purpose of this reduction is to increase the rate of `Feedback` message and thus speed up the acknowledgement of feedback and free the memory used to store the cache. The details of this will be explained in the Section 4.6.

4.5 Multipath integration

The multipath part of the implementation requires some modifications at some strategic points: selection of a flow to send a packet, reception from all the subflows simultaneously,... We also need to store a few structures in memory to keep subflows state and statistics. This section aims to cover all these modifications, that make multipath possible in wolfSSL.

4.5.1 Memory structures

To be able to use multipath, we needed to store some information in the memory, like the addresses the application wants to advertise, the addresses of the remote host, the created subflows and their statistics, the subflows waiting for confirmation and the sockets opened to reserve port numbers but not actively used in a subflow. We have implemented helper functions to handle these structures easily, such as inserting new elements, removing element either by index or with some criteria and searching for elements. These functions take care of reallocating the memory correctly in order to keep the memory footprint as low as possible.

```
typedef struct MPDTLS_ADDRS {
    struct sockaddr_storage* addrs;     /* Contains all the available addresses
                                         and ports for MPDTLS (both IPV4 or IPV6) */
    int                      nbrAddrs; /* Number of available addresses */
} MPDTLS_ADDRS;
```

Listing 4.1: Structure storing addresses

In our implementation, we have added two `MPDTLS_ADDRS` structures to the general `WOLFSSL` *session object* and one to the `WOLFSSL_CTX` *context object*.

The first structure `mpdtls_host` contains the addresses advertised by the host. An application can insert new addresses into this list by two ways; first, it can call the `wolfSSL_mpdtls_new_addr (WOLFSSL*, const char*)` function, with the session object and an address as a string, under

either IPv4, IPv6 or DNS format, as arguments. In case of DNS, all addresses found during the resolution are added at once. The other way is to call `wolfSSL_mpdtls_new_addr_CTX (WOLFSSL_CTX*, const char*)`. This contextual variant adds the address in the *context object* rather than in the *session object*. All the sessions created from this context will inherit its host addresses. To ensure that this list of registered addresses is correctly synchronised, a `CIM` is issued either each time a new address is added or just after the handshake of a session inheriting addresses from its context.

The second structure `mpdtls_remote` stores the addresses advertised by the other host. It is updated only after reception of a `ChangeInterfaceMessage` and is used to know which subflows can be created.

```
typedef struct MPDTLS_SOCKS {
    int* socks;    /* Contains all the available sockets for MPDTLS */
    int  nbrSocks; /* Number of available sockets */
} MPDTLS_SOCKS;
```

Listing 4.2: Structure containing inactive socket

However, from the Section 3.1.2 and more specifically the Listing 3.2, we can see that each address advertised must also mention the port reachable for the flow. As the subflow does not already exist, the port is still unknown when advertising. To determine this port, we let the OS assign a port number to the application simply by opening a socket. We then retrieve the port number and register it in the `MPDTLS_ADDRS` structure. But, if we delete the socket just after, the port number could be reassigned, and thus would become unavailable for the library. To avoid this issue, we store the created socket in a `MPDTLS_SOCKS` structure (Listing 4.2). When we finally create a subflow with the address corresponding to a socket of this pool, we reuse it instead of creating a new socket. In this way, we keep the amount of open sockets low while keeping the advertised port number safe.

```
typedef struct MPDTLS_FLOW_HEARTBEAT {
    struct timeval last_heartbeat;         /* last timestamp */
    byte           response_rcvd;          /* whether or not we have received
                                              a response to our heartbeat */
    uint           rtx_threshold;          /* the retransmission threshold
                                              will be higher if no response */
    byte*          heartbeatPayload;       /* heartbeat payload */
    word16         heartbeatPayloadLength; /* length */
    MessageState   heartbeatState;         /* avoid multiple heartbeat tx */
} MPDTLS_FLOW_HEARTBEAT;

typedef struct MPDTLS_FLOW {
    struct sockaddr_storage host;           /* flow determined by the host */
    struct sockaddr_storage remote;         /* & remote sockaddr (ip+port) */
    int                     sock;           /* connected socket (if any) */
    uint                    wantConnectSeq; /* last wantConnect sent */
    uint                    tokens;         /* tokens count of this flow */
    MPDTLS_FLOW_HEARTBEAT   hb;             /* hb manager */
    MPDTLS_SENDER_STATS     s_stats;        /* stats for sent packets */
    MPDTLS_RECEIVER_STATS   r_stats;        /* stats for received packets */
} MPDTLS_FLOW;
```

```
typedef struct MPDTLS_FLOWS {
    int           nbrFlows;      /* Number of available flow */
    int           cur_flow_idx;  /* Flow selected for sending */
    uint          token_counter; /* total number of tokens */
    MPDTLS_FLOW* flows;          /* collection of flow */
} MPDTLS_FLOWS;
```

Listing 4.3: Structures handling flows

4.5.2 Subflow creation

Following the exchange of `WantConnect` and `WantConnectAck` (explained in Section 3.2), a new subflow can be opened. From an implementation viewpoint, it means that this new subflow must be added in the `MPTDLS_FLOWS` structure (shown on the Listing 4.3) and a new socket must be opened and connected with the other host address and port.

However, as seen in the Section 3.2, the host that receives a connection request has to set up its socket and flow structure before having the confirmation that the subflow must be created. As the `WantConnectAck` can be lost on the return path, it is not impossible the request issuer asks again the opening of the same connection. Therefore, we needed a way to differentiate well-established subflows and subflows requiring confirmation (realized by a `Heartbeat` exchange).

This differentiation is made through the use of a second `MPDTLS_FLOWS` structure in memory, containing the waiting flows. This way, we know if a request is legitimate or if it is a replay due to a loss occurred during the process, simply by checking the existence of the requested subflow into this secondary structure. To avoid an unlimited growth of this structure, we check in the `CheckTimeouts` it is not in waiting state for too long, otherwise we forget the corresponding request.

To detect these waiting subflows, we use the existing `last_heartbeat` field to store their creation time. In the current implementation, the timeout for the waiting subflows is defined through a preprocessor constant and set to 300 seconds.

Otherwise, if a heartbeat request is received on a waiting subflow, this latter is immediately considered as active and transferred to the primary structure.

4.5.3 Multipath I/O

Once we have multiple subflows, we need to listen to all of them. In the original implementation of WolfSSL (explained in Section 4.2.4 and illustrated in Figure 4.1), the library directly listened to the flow by calling the `CBIORecv` macro with the socket descriptor, with CBIORecv generally pointing to `recvfrom` on a Unix system. To listen to all the registered sockets, we cannot directly go into `CBIORecv`. Instead, we can use the function `select(2)` to listen for incoming packets on all the sockets, and once a packet arrives, we can call `CBIORecv` with the socket returned by `select`. So, we can see the implementation of the receiving part of Multipath DTLS is quite easy.

The sending part (Figure 4.2) is even easier, as we just need to replace the socket used to send the current packet through `CBIOSend`. This replacement is made in `SendBuffered` to disturb as few as possible the library. As DTLS already used the field `ssl->buffers.dtlsCtx.fd` to indicate the socket descriptor to give to `sendto`[3], we reuse this field. There are two possibilities to determine the socket descriptor to put in the `dtlsCtx`: the path manager (explained in the subsection 4.5.5) or the preference setting.

Sometimes, we need to explicitly specify the flow to use for a message, like for the `Heartbeat Response`, which must go back on the same flow as the `HeartbeatRequest`. For this, we added a special option in the session object allowing us to specify a preferred flow. If this option is set, it has priority on the Path manager and the packets will be sent on this socket, without asking to the path manager which flow should be used.

4.5.4 Failure detection

In case of a link failure, we must delete the subflow which has failed to redirect the traffic to other available subflows. This is done in the `SendTo` and `ReceiveFrom` functions by treating the errors reported by the socket. If an interface fails because it has been removed from the computer (e.g. Wi-Fi USB stick) then the socket will return an error and we will loose the current packet. But all the following packets will be routed to other subflows. Similar behavior happens if we bring our smartphone out of the Wi-Fi range: an `address unreachable` error will be reported.

4.5.5 Path manager

To compute the fractional distribution, the `apply_scheduling_policy()` function is called every time we receive a feedback containing new information about the links. Different scheduler policies may be implemented but the principle is always the same: to give more or less weight to a subflow. To do so, each subflow has tokens and the ratio $\frac{tokens}{total\,tokens}$ gives the fraction of the traffic going through this subflow. We have added a new API call to easily change the scheduling policy at any moment the communication. More information about the different scheduling policies are given in the Section 5.2.

In addition, we have implemented 2 schedulers that are independent of the scheduling policy. Their role is to select a subflow for a particular packet given a fractional distribution. The first option explored is to build a deterministic scheduler that will iterate sequentially over the different subflows. Thanks to the tokens approach, the complexity of this scheduler is of $O(1)$. The second option is to introduce some randomness with a non-deterministic scheduler. Before sending a packet, a new random number is generated between 1 and the total number of tokens. The corresponding subflow is chosen to handle the packet. This second approach is probably better because it is not likely to produce burst of packets as the first one. Unfortunately to determine the subflow which corresponds to the random number generated we need $O(n)$ instructions where n is the number of subflows. Although it may be possible to reduce

[3] which is generally the function called by the `CBIOSend` macro on a Unix system

this complexity with advanced data structures such as HashMap, we don't have explored these options yet.

Since we only provide examples of schedulers and an application may need to have its own scheduler to answer specific needs, we give a way to change it easily. The `wolfSSL_CTX_SetScheduler(WOLFSSL_CTX*, CallbackSchedule)` function will give a way to an application to specify in the *context object* which scheduler should be called every time we want to send a packet. This function will take the *ssl object* in parameter as well as our `MPDTLS_FLOWS` structures to access all data about each subflow and be able to make a choice.

4.6 Compute the statistics

To compute the statistics on different paths, we have followed the concepts we explain in Section 3.3. Every connection can be considered as two half-connections, each of them implying a sender and a receiver. In the following sections, we present what we have implemented and review some concrete examples.

4.6.1 Sender side

Structure in memory

The structure we use to store the sender statistics is presented on Listing 4.4. One of the design choice is to store the sequence number in a structure similar to the `ArrayList` of Java[4]. Indeed, we have an array with a defined capacity and we can add as many elements as we want. To avoid too many memory allocations, we always double the size when we need more space. In consequence, the variable `capacity` remembers the real size of the array in memory while the `nbr_packets_sent` indicates the number of elements stored in the array. A variable size array is needed because we never know in advance how many elements will be stored. It will depend on the lost rate of feedback packets.

```
typedef struct MPDTLS_SENDER_STATS {
  uint*    packets_sent;     /* sequence number of packets sent */
  uint     capacity;         /* capacity of the array (mimic arraylist) */
  uint     nbr_packets_sent; /* number of stored packets inside packets_sent */
  uint     waiting_ack;      /* first packet which has not been transmitted */
  uint64_t forward_delay;    /* average forward delay (ms) */
  float    loss_rate;        /* loss rate computed */
} MPDTLS_SENDER_STATS;
```

Listing 4.4: Sender structure to store statistics

[4] https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

Processing feedback

The sender must keep in memory an array of the packets sent on each path. Nevertheless this list cannot grow infinitely as the sender is supposed to receive regularly some feedback from the receiver. The way each feedback will impact the list is depicted on Figure 4.3. For the sake of simplicity, we have only considered 3 fields of the feedback packet, namely the feedback sequence number, the minimum and the maximum sequence number received. At the beginning, no feedback at all has been received and all packets have the same status. Then a first feedback packet is received and reports an acknowledgement for the packets 1 to 4. These particular packets obtain a special status, they have been reported a first time but they still can't be removed from the list. Indeed, we never know at this stage if the feedback ack has been received successfully by the other side. This separation is assured by the `waiting_ack` variable from Listing 4.4 which always remembers the position of the first packet not reported yet (5 in this case).

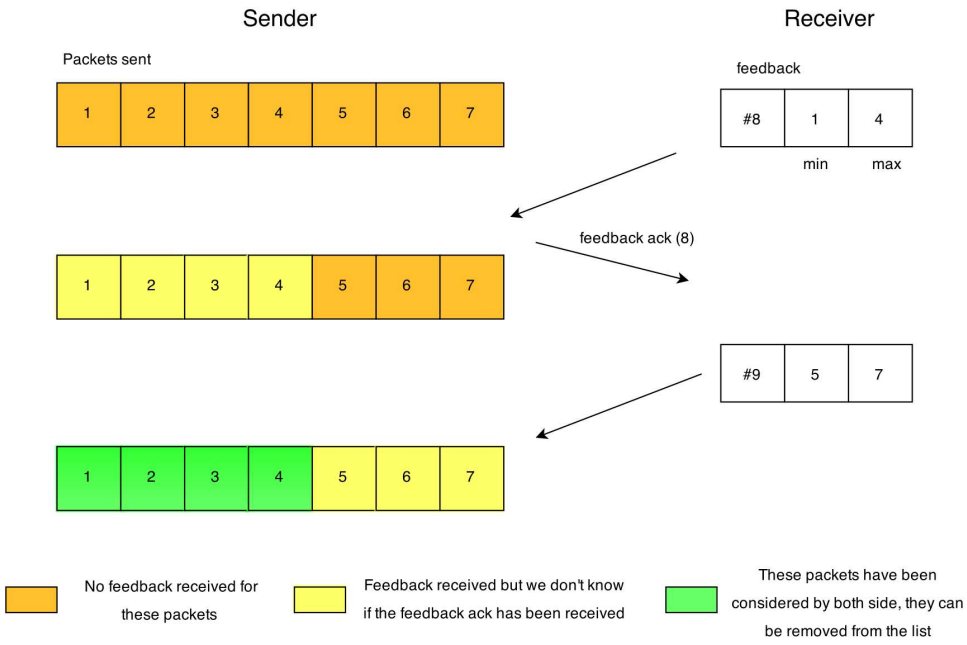

Figure 4.3: How the feedback works in the sender side

When a second feedback arrives, everything below the minimum sequence number reported (packets 1 to 4 in this case) can be removed from the list, because the feedback packets are cumulative. It will probably become clearer with the second example on Figure 4.4. It is exactly the same scenario as before except that the `FeedbackAck` is lost. Therefore, on the next feedback, the receiver will report all the statistics from the beginning. In consequence, the sender will have to wait one more feedback to shrink the list.

Note that in our example the sequence numbers are sequential. It is not a requirement since in practice packets are sent on different paths and therefore some gaps may be present between sequence numbers on a particular path. Nevertheless the sequence number must remain strictly increasing over time for the max-min mechanism to work well. This is guaranteed by DTLS and even if we reach the maximum sequence number allowed, the handshake must be first repeated according to RFC5246[1].

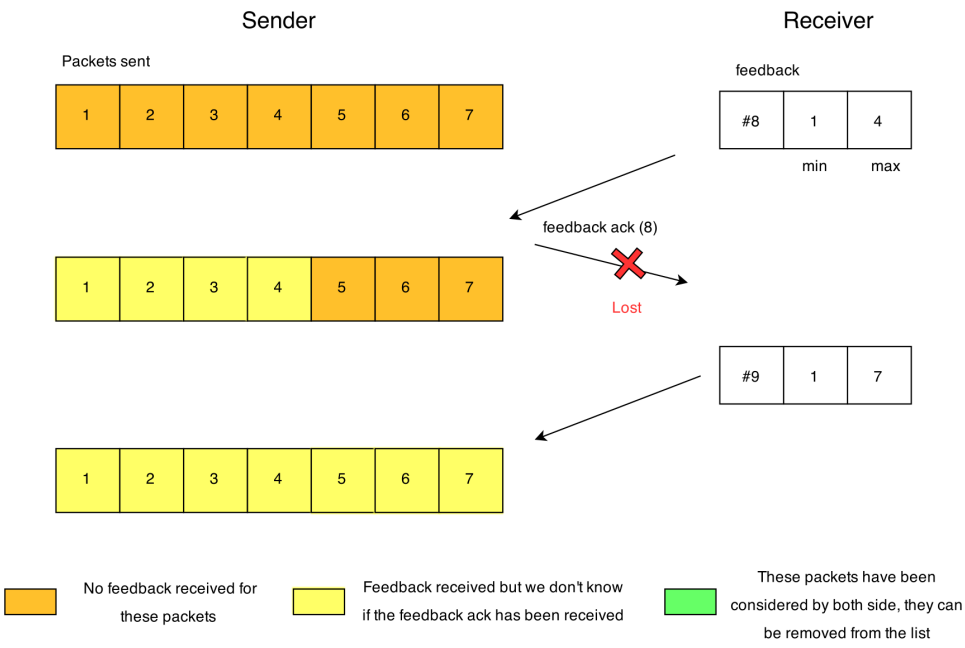

Figure 4.4: How the feedback works in the sender side with losses

The loss rate is computed every time we receive a feedback on the range [min,max] carried by this feedback. It sometimes implies to compute the loss rate on overlapping sets of packets. This is the case with the scenario presented on figure 4.4: we first compute the loss rate on the packets 1 to 4 and then a second time on the packets 1 to 7. As stated in the design part, the loss rate is only an estimation of the real loss rate. However in practice, our system estimates quite accurately the real loss rate (see Section 5.3.2 for more details). In addition, as recommended in Section 3.3.2, we use an exponential average to consider new loss rates. This prevents short term statistics to impact too heavily the global loss rate of the subflow.

4.6.2 Receiver Side

Structure in Memory

As for the sender statistics, each subflow has to keep some information in memory to monitor the quality of the link. The C structure used is presented on Figure 4.5. The biggest difference with the sender is the constant size of the structure. Indeed, we don't need to remember every packet received but only the minimum and maximum sequence number received so far.

```
typedef struct MPDTLS_RECEIVER_STATS {
  /* information gathered since the last feedback */
  uint64_t nbr_packets_received;   /* number of packets received */
  uint     min_seq;                /* minimum sequence number */
  uint     max_seq;                /* maximum sequence number  */

  /* same information as before but transmitted */
  uint64_t nbr_packets_received_cache;
```

```
  uint      min_seq_cache;
  uint      max_seq_cache;

  uint64_t backward_delay; /* average backward delay (ms) */
  int       threshold;      /* after how many packets must we send a feedback */
  uint      last_feedback;  /* sequence number of the last feedback we sent */
} MPDTLS_RECEIVER_STATS;
```

Listing 4.5: Receiver structure to store statistics

Returning feedback

The way the feedback is handled on the receiver side is simpler than on the sender's. We just need to support cumulative feedback. Figure 4.5 presents the same scenario as Figure 4.4 but on the receiver side. The feedback ack being lost, packets 1 to 4 are not removed. Indeed, the receiver cannot make the distinction between the loss of the original feedback and the feedback ack one.

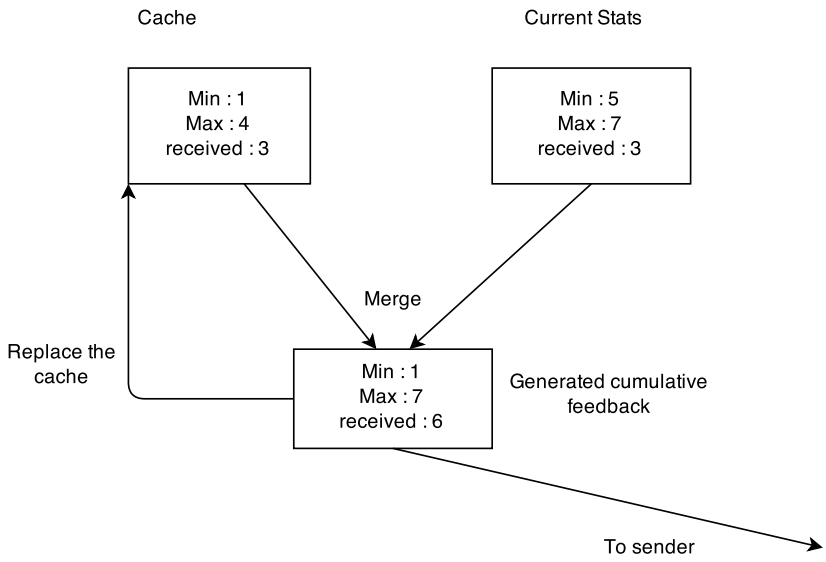

Figure 4.5: How the feedback is generated on the receiver side

What we have called the "cache" will aggregate all previous feedback. When the threshold is reached and the receiver has to send feedback, it first merges the current statistics with the cache to generate the cumulative feedback. The latter is sent to the sender (on the same flow) and the sequence number is stored in the `last_feedback` variable of Listing 4.5. Then the cache is replaced by this new generated feedback.

Processing feedback ack

When a feedback ack is received, the receiver must check if the sequence number corresponds to the last feedback sent. If they match then the cache is emptied and we are in a situation similar to the one depicted

on Figure 4.3. The feedback ack carries the sequence number 8 which matches the latest feedback sent thus the cache containing packets 1 to 4 is emptied. Therefore the next feedback will report statistics on packets 5 to 7 only.

If the feedback ack does not contain the sequence number of the latest feedback sent we simply discard it. This kind of situations may be encountered if we have packet interleaving.

Computing backward delay

The backward delay is computed thanks to regular heartbeat messages going from the sender to the receiver. These messages must transport a timestamp so we have defined a new type of heartbeat called `HEARTBEAT_TIMESTAMP`. It allows us to differentiate this kind of message at the record layer stage and perform a different treatment than on standard `HEARTBEAT_REQUEST`.

To get the current time of a system, we use the `gettimeofday()` function with a `timeval` structure. It gives a microsecond precision but we actually decided to consider only the milliseconds. If there are two links we cannot differentiate at the millisecond level, we can just consider them as of equal speed (i.e. the gain will not be significant enough to treat them differently). Of course this could be adapted easily if we have a real interest for such precision and if network components become much faster.

When the receiver treats a `HEARTBEAT_TIMESTAMP`, it computes the absolute value of the difference between his current time and the one contained in the message. This will give the backward delay with the clock desynchronization term. The receiver will add these delays and keep in memory the exponential average (variable `backward_delay` of Listing 4.5). Every time it sends a feedback, it includes its estimation of the backward delay which gives the sender its forward delay.

5 | Performance Evaluation

5.1 Test application

As MPDTLS is implemented inside a library, we need an application to use it. We tried to find out an existing application that could be a good candidate for our tests. First, we needed an application which uses DTLS for most of its communications. It was surprisingly hard to find but we still managed to find one : Campagnol[43]. Campagnol is a decentralized VPN solution that uses DTLS to communicate between the peers. Unfortunately, for reasons exposed in Appendix A, we were unable to integrate correctly our modified library within this application.

So, we took the decision to build a simple VPN application [44] by reusing some part of Campagnol code. A VPN application is indeed perfectly suitable for all the experiences we could imagine : potentially any application can use a VPN tunnel without even noticing it. Each packet is encapsulated inside a DTLS packet transmitted securely between the two MPDTLS hosts (see Figure 5.1).

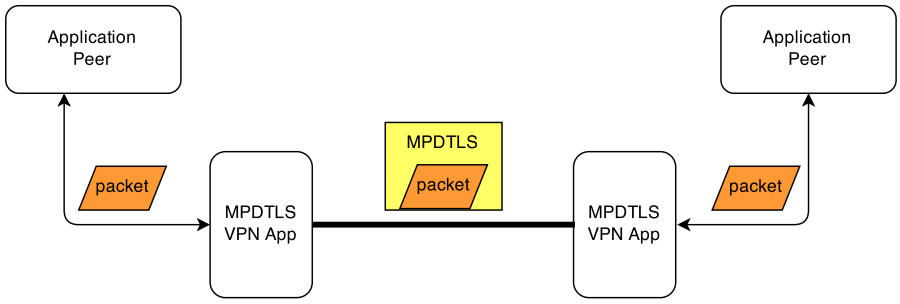

Figure 5.1: A simple VPN application using MPDTLS

The behavior of such a MPDTLS VPN App is shown on Figure 5.2. A TUN interface is created with a specified address and netmask. Every packet going to an address falling in this netmask is sent by the kernel to this interface. Then we need to monitor this interface, to read the incoming packets and to sent them through the tunnel. This is the job of the first thread (in red on the Figure). We also need to capture packets coming from the network and to forward them to the TUN interface. This is handled by a second thread represented in green on the Figure. Finally, a last thread (in blue) listens to the standard input for potential commands. This is the channel that we use to communicate with the application to add or remove new interfaces for instance. We can ask for debug information or even change the scheduler policy on the fly.

One of the most useful information we can ask for are the statistics for a particular flow. Looking at this output, we can identify most of the time an issue without analyzing at the packet trace. An example of such an output is shown on Listing 5.1. We can easily see the part of the traffic that the subflow is

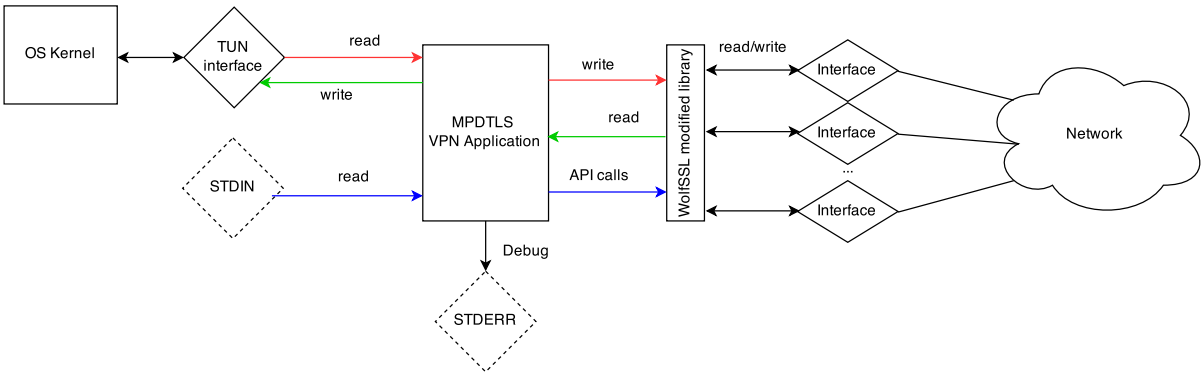

Figure 5.2: I/O interactions for one host of our application. Each thread is represented with one color.

actually supporting, the estimated delays and other information. The separator "|" used in the `packets sent` list differentiates the ones that are in the waiting status and the others (see Section 4.6 for more details about this waiting status).

```
---- Stats Flow N 0 ----
IP src : 11.0.0.1
IP dst : 11.2.0.1
Support 41 % of the connection
----- Receiver Stats -----
Packets received : 44
Min_Seq received : 76
Max_Seq received : 172
Backward delay : 10 ms
----- Receiver Cache -----
Packets received : 0
Min_Seq received : 2147483647
Max_Seq received : 0
----- Sender Stats -----
Packets sent : [ 8487 8489 8492 ... 8633 8634 8636 | 8640 8642 ... 8706 8707]
Forward delay : 10 ms
Loss Rate : 0.000000
--------------------------
```

Listing 5.1: An output of the statistics for a particular flow

5.2 Different scheduler policies

We present here three examples of scheduler strategies that we have implemented and how the fractional distribution is computed in each case. As conclusion, each of them has pros and cons. The good choice is strongly related with what the underlying application expects to do. In some cases, an application could want to build its own scheduler and this matches with our design since we provide a dedicated customisable callback.

5.2.1 `Round Robin`

The round robin is the simplest strategy consisting in sending the same amount of packets over each available link. This does not use any of the information gathered from the feedback. However it may be a good choice if we know in advance that the different links share common properties (e.g. in a datacenter).

The fractional distribution is really easy to compute as every link will get the same fraction of the traffic (in terms of number of packets).

$$f_i = \frac{1}{n}$$

where f_i is the portion of traffic given to flow i and n is the total number of subflows.

5.2.2 `Optimize Latency`

For this strategy, we want to give more weight to the links where the forward delay is lower. We have to keep in mind that every delay d_i contains the clock desynchronization term ΔT (see Section 3.3.1). In order to get rid of this term, we must consider the difference with another delay. We choose to take the difference with the maximum delay as by definition the greater the difference, the smaller the delay. This maximum will evolve and is defined as the maximum delay reported on every flow at the time we compute this fractional distribution. Equation 5.1 gives the complete expression.

$$f_i = \frac{max_d - d_i + \alpha}{\sum_j (max_d - d_j) + n * \alpha} \quad \text{where} \quad max_d = max_i(d_i) \tag{5.1}$$

Without the α term, the flow with the maximum delay will never be used. Indeed if you have two subflows with forward delays of 10ms and 20ms and $\alpha = 0$, then 100% of the traffic will be supported by the first subflow. Although this could appear as an optimal choice, it is always preferable to send a small part of the traffic on each link just to monitor the characteristics and to keep receiving feedback and heartbeat messages. After some tests, we think the value of α must be defined between 5% and 10% of max_d[1] to give the best results.

5.2.3 `Optimize Loss`

Another strategy that we could consider is to favor the less congested links. In this case, we give more priority to a link with the smallest loss rate. The principle is almost the same as for the latency, we consider the difference with the biggest loss rate observed. The idea is to quantify the relative differences between the subflows.

[1] Of course, max_d must not be null. If it is the case, then a constant value must be chosen like 1ms and the equation will actually give the same result as a round robin.

We may be tempted to use directly the loss rate reported in the statistics. However if the link experiences severe losses, we may not be aware of it because all the feedback packets have been dropped. Therefore we compute what we call the "real" loss rate which also takes in consideration the number of packets sent and not yet acknowledged. Equation 5.2 gives the formula used and $feedback_{thr}$ is the number of packets after which we send a feedback. We consider two times this amount to give a penalty to the link because after one feedback, packets are in waiting status and need another feedback before being completely forgotten. The penalty pen is computed as two times the probability of loosing one feedback. The factor 2 here will probably need some tuning but the idea is to boost the penalty as the risk is rather small to loose only the feedback packet but no application data.

$$LR_i = stats_{LR} + \left\lfloor \frac{pckts_{sent}}{2 * feedback_{thr}} \right\rfloor * pen \qquad \text{where} \quad pen = \frac{1}{feedback_{thr}} * 2 \tag{5.2}$$

$$f_i = \frac{max_{LR} - LR_i + \beta}{\sum_j (max_{LR} - LR_j) + n * \beta} \qquad \text{where} \quad max_{LR} = max_i(LR_i) \tag{5.3}$$

Equation 5.3 is really similar to Equation 5.1 and we also need a constant term β for the same reasons. After some tests, we arrive at the conclusion that a suitable value for β is around 1% of the loss rate in most situations.

5.3 Multipath simulations

We have designed an environment to evaluate our system under different conditions. All the following measures take place inside a Mininet laboratory [45] to easily set up the topology we want and to create multiple interfaces. Also, we used a kind of framework for Mininet called Minitopo[46] to easily configure the topology and create automated tests just with configuration files instead of scripts. To generate network traffic, we used the D-ITG (Distributed Internet Traffic Generator)[47] tool. More details about how to reproduce these graphs and the following are given in Appendix C.

5.3.1 Topology

The topology used for the following evaluations is the one presented on Figure 5.3. We have four interfaces on the client side that are linked to a network router and one link between the router and the server. Note that the latter is not constrained, we will only change the characteristics of the first 3 paths shown on the Figure. The path 4 is reserved for the signaling of D-ITG to avoid any interference with our measures.

The corresponding logical topology is depicted on Figure 5.4. We are using 3 flows concurrently between the client and the server. The D-ITG application is sending traffic to one unique TUN interface and the traffic goes through our tunnel to reach the D-ITG receiver.

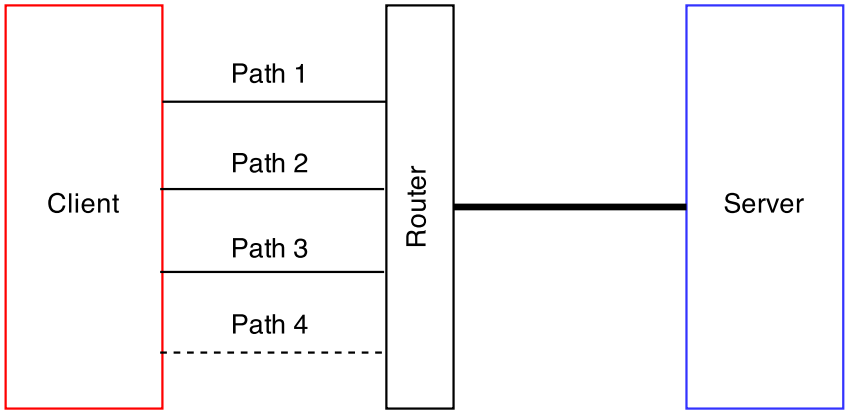

Figure 5.3: Physical topology inside mininet

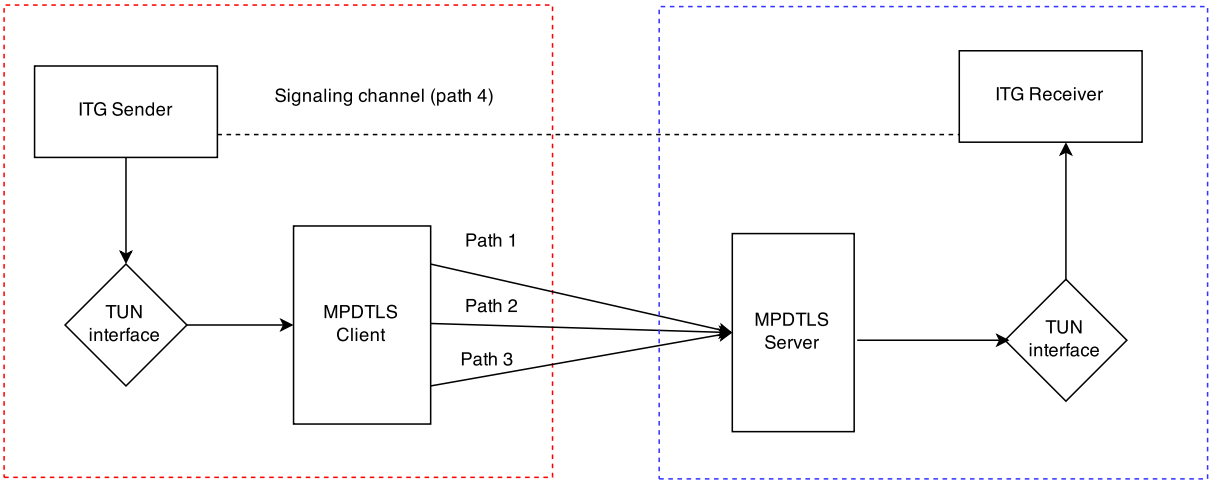

Figure 5.4: Logical topology used for measurements

5.3.2 Traffic balancing

The first thing to evaluate is if we really take advantage of multiple interfaces and how we balance the traffic between them. We performed experiments to determine in which conditions each scheduler works well and how they compute the distribution of traffic in various environments.

With dynamic bandwidth

In this experiment, we have tested the two schedulers that are taking the context into consideration : `Optimize Latency` and `Optimize Loss`. The configuration used is the following (the path indexes refer to Figure 5.4) :

Path n°	bandwidth	loss rate	delay
1	1 Mbps	0	10ms
2	variable	0	20ms
3	not used	-	-

We ran the experiment with multiple values of bandwidth of path 2 to see how its usage rate is impacted. We have generated constant traffic with D-ITG [47] using both UDP and TCP. All these results are reported in Figure 5.5. The measurements have been obtained with sessions of 60 seconds and every experiment has been repeated 11 times to evaluate the mean. The 95% confidence interval of the mean is presented in white and has been computed with a Student T distribution.

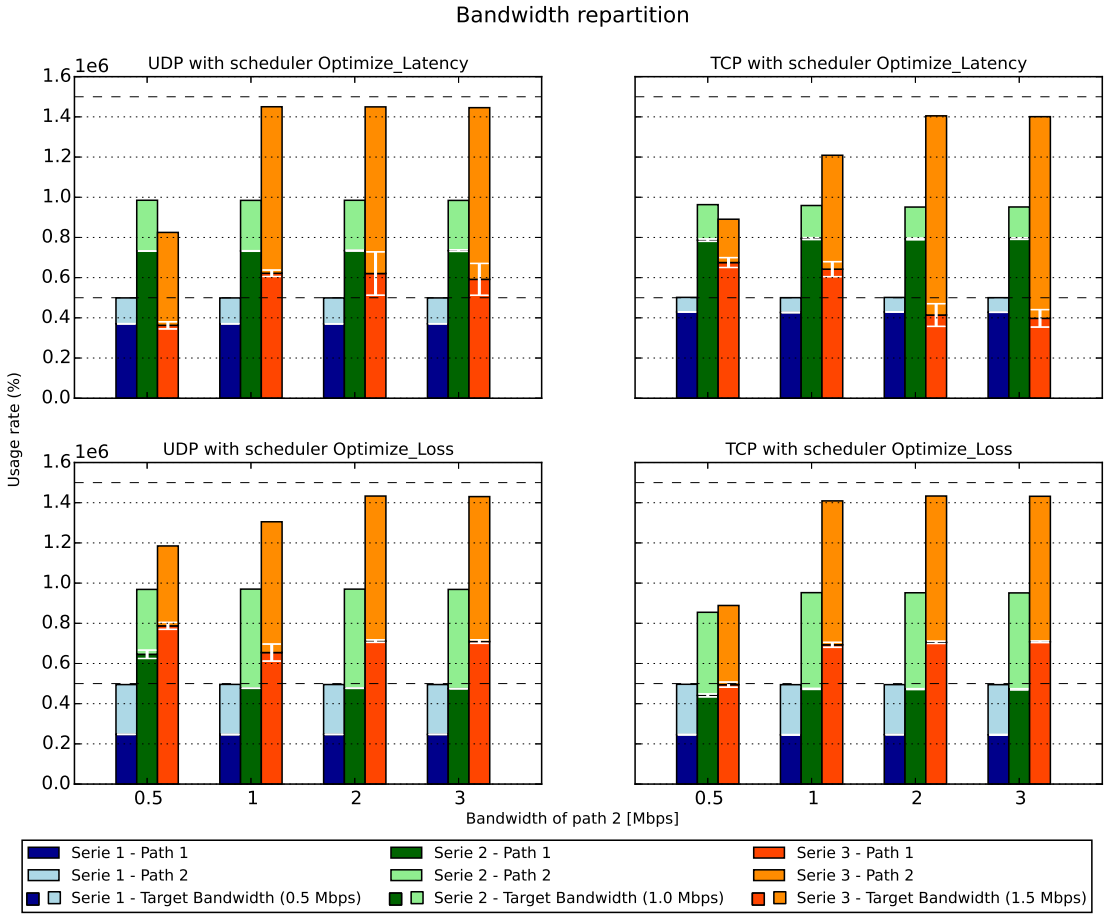

Figure 5.5: Usage repartition with bandwidth variation

For every bandwidth value of path 2, we have 3 bars that represent the total traffic we are trying to send through the tunnel. Each bar is separated in two parts that show the amount of traffic passing through each path in bytes/sec. The losses are easily identifiable because they occur every time the total usage does not reach the total bandwidth. Of course at the beginning we have congestion because if path 2 can support only $0.5Mbps$ then $bw(p1) + bw(p2) = 1.5Mbps$ and in the last case we are trying to send traffic at 1.5Mbps. So, it is supposed to use 100% of the tunnel capacity which is impossible in practice. Moreover, this computation does not even take into account the overhead introduced by the encapsulation inside DTLS. Indeed, we are sending packets of size 825 bytes into the TUN interface and the resulting size of the DTLS `ApplicationData` packet is 905 bytes; so an overhead of 80 bytes (almost 10% in this case).

First, it is interesting to confirm that the `Optimize loss` scheduler performs better if we are dealing with congested links and especially with UDP. Note that TCP doesn't experience losses in the same way

as UDP because it will reduce its transmission rate internally with the congestion window mechanism. So even if we impose a given volume of traffic, the real bandwidth will be lower with TCP in case of congestion.

Secondly, we observe an unexpected phenomenon if we look at the graph "UDP with scheduler Optimize Latency" : when we increase the total bandwidth to 1.5Mbps (orange), it does not optimize latency at all. Indeed, it gives only a small part of the traffic to the faster link. We have tracked down the problem and it is apparently coming from a bad perception of the forward delay. At the beginning, it uses extensively path 1 as expected but rapidly reaches congestion as the capacity of path 1 is 1Mbps. The heartbeat messages that are used to evaluate the forward delay are themselves postponed by a few milliseconds due to this congestion and it is sufficient for the scheduler to estimate the delay is better on path 2. Even if the forward delay is later correctly estimated, we go back in the same cycle. It is not the case with TCP at least at the beginning because TCP reduces the bandwidth to fit into the tunnel and therefore there is no congestion observed.

With dynamic loss rate

This time, we want to see how the schedulers react with various loss rate at one of the paths. The configuration is the following :

Path n°	bandwidth	loss rate	delay
1	1 Mbps	0	10ms
2	3 Mbps	variable	10ms
3	not used	-	-

Again we are using 2 paths only and they have the same latency but different bandwidths. We run different experiments with 4 values of loss rate for path 2, going from 0% to 3%. The results we have obtained are presented on Figure 5.6.

A first element to point out is the fact the `Optimize Latency` scheduler is not really impacted by the loss rate as we could expect. The proportion is always around 50% for both TCP and UDP as the two links have the same delay. The little exception to this rule is noticed for the third bar (traffic at 1.5Mbps) probably because we reach the congestion limit on path 1. Therefore, the heartbeat messages are either delayed or dropped and the forward delay is overestimated.

On the other side, the `Optimize loss` scheduler will progressively give more weight to the first path as it is loss free. We state that UDP and TCP give different results concerning the last bar which corresponds to the 1.5 Mbps transfer speed. Again, this can be explained by the congestion on path 1. UDP keeps sending packets even if they are dropped, therefore the scheduler will detect important losses on path 1. The loss rate caused by congestion will overcome the one of path 2 and the scheduler will thus choose the best option among the two. However, for TCP, the congestion control mechanism will reduce the sending rate to avoid congestion of path 1. In such conditions, path 1 remains the best option and thus receives more weight from the scheduler.

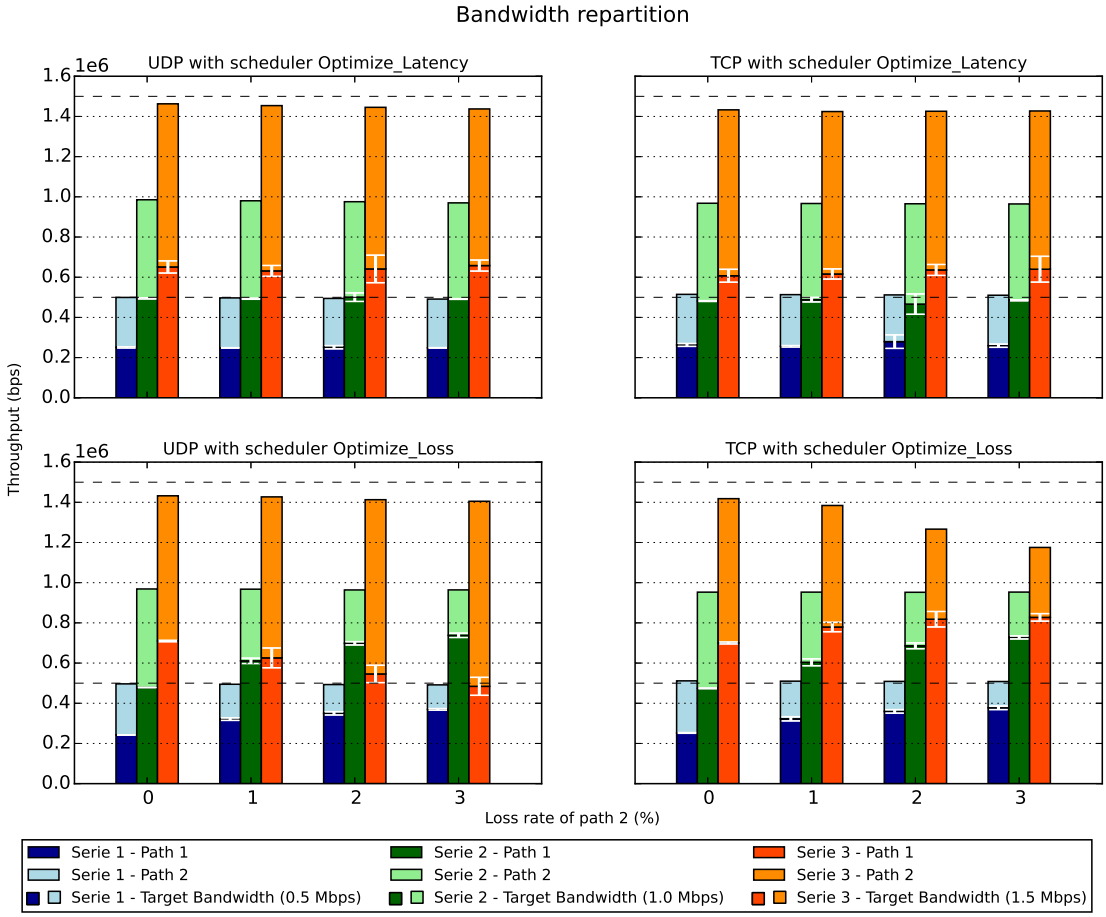

Figure 5.6: Usage repartition with loss rate variation

5.3.3 Resiliency to interface removal

The objective of using multipath is not only to balance the traffic on the existing paths but also to modify dynamically the distribution of the traffic if one interface becomes unavailable. For this experiment, we use the topology of Figure 5.4 with the following configuration :

Path n°	bandwidth	loss rate	delay
1	5 Mbps	0	10ms
2	5 Mbps	0	20ms
3	5 Mbps	0	40ms

We generate constant UDP traffic at 3 Mbps and observe the behavior of the implementation if path 1 is suddenly broken at $t = 19s$. The bandwidth repartition is shown on Figure 5.7 and the scheduler used is `optimize loss`.

Note that the path 1 was used for the handshake and we prove here that even if we remove our "primary" interface, the connection continues. In the very first moments (at 4s), we note that the distribution is more or less equal among the subflows. This setup time is needed to obtain some information about the forward delay. Heartbeat messages and feedback packets have to be exchanged between the two hosts.

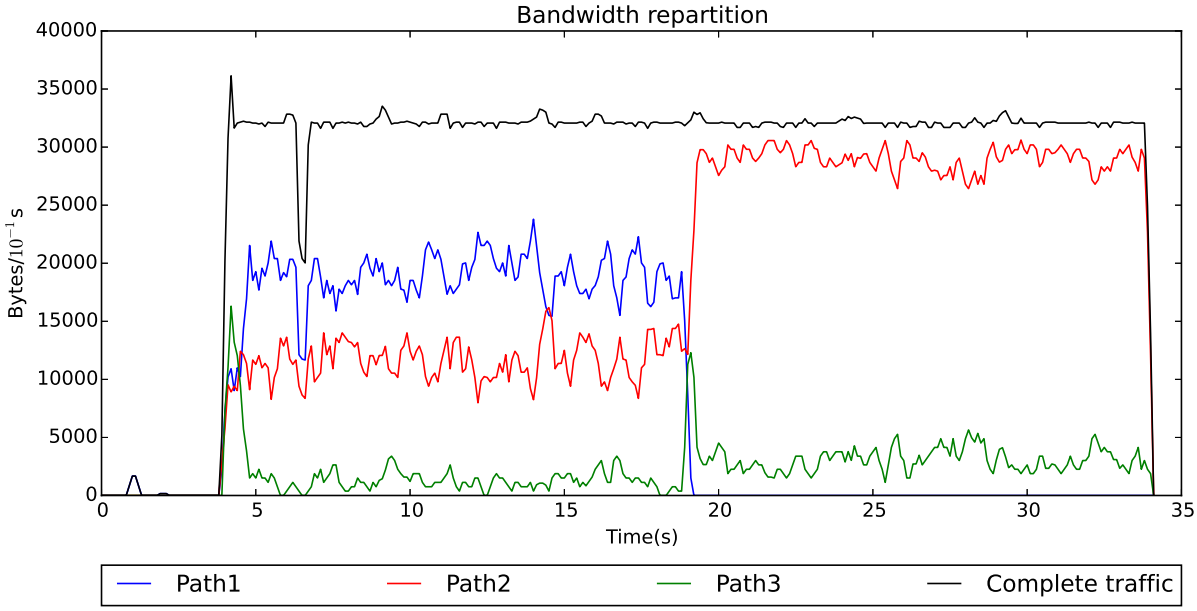

Figure 5.7: Reaction to interface removal

After 5 seconds, we see that the distribution is shaped with what we can expect from Equation 5.1: path 1 has almost $2/3$ of the traffic because it has the best delay, path 2 is still used because the difference of delays with path 1 is acceptable and then some traffic is given to path 3 to monitor the link. Of course, a more aggressive scheduler will probably let path 1 support 100% of the traffic but the objective here was to use all paths to perform probing.

When path 1 fails, the distribution is recomputed and most of the traffic is re-routed to path 2. Of course we are far from the congestion because each link has a bandwidth of 5 Mbps but the objective here was to see how the distribution is moving without other constraints. In this context, the scheduler is doing well by optimizing the overall latency as we can see in Figure 5.8. After the loss of the fastest path, the overall delay is increased and is little above the 20ms threshold which is the delay of the new fastest link. We also can observe two peaks: the first one is due to the setup time and the second one is a temporary increase of path 3 usage to compensate the failure of path 1.

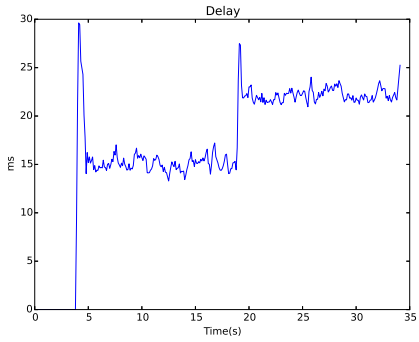

Figure 5.8: Overall delay

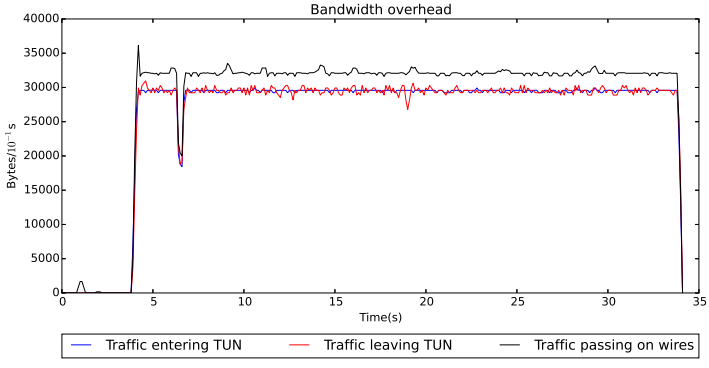

Figure 5.9: Application perception of traffic

From the application's point of view, the impact of such failures is shown in Figure 5.9. We see in blue

the traffic entering the tunnel which is constant here because we have parametrized D-ITG to do so, in red we see the traffic leaving the tunnel and in black we have the traffic generated by the tunnel. The difference between the blue and the black curves is the overhead caused by the encapsulation of D-ITG packets inside DTLS ones. Around the 19^{th} second, we notice a really small drop on the red curve, exactly when the interface was lost. This is the only thing the application will perceive from the loss of one interface. This is a huge improvement in comparison with a normal DTLS connection, which would have ended the communication in case of such interface loss.

5.3.4 Smooth addition of a new interface

In this section, we explore the reverse scenario: when one interface becomes available. At the beginning only path 1 and path 2 are available and at time $t = 30s$ we add path 3. The configuration used is the following :

Path n°	bandwidth	loss rate	delay
1	5 Mbps	0	30ms
2	5 Mbps	0	40ms
3	5 Mbps	0	10ms

As we can see path 3 has the lowest delay and we expect that it will take the lead over the two other ones. This is verified with our experiment as we can see on Figure 5.10. A significant portion of the traffic is redirected through the new flow. At the beginning path 1 was the fastest link so it was given the largest part. But after the addition of the new interface, a re-computation is made according to equation 5.1 and path 1 is less used.

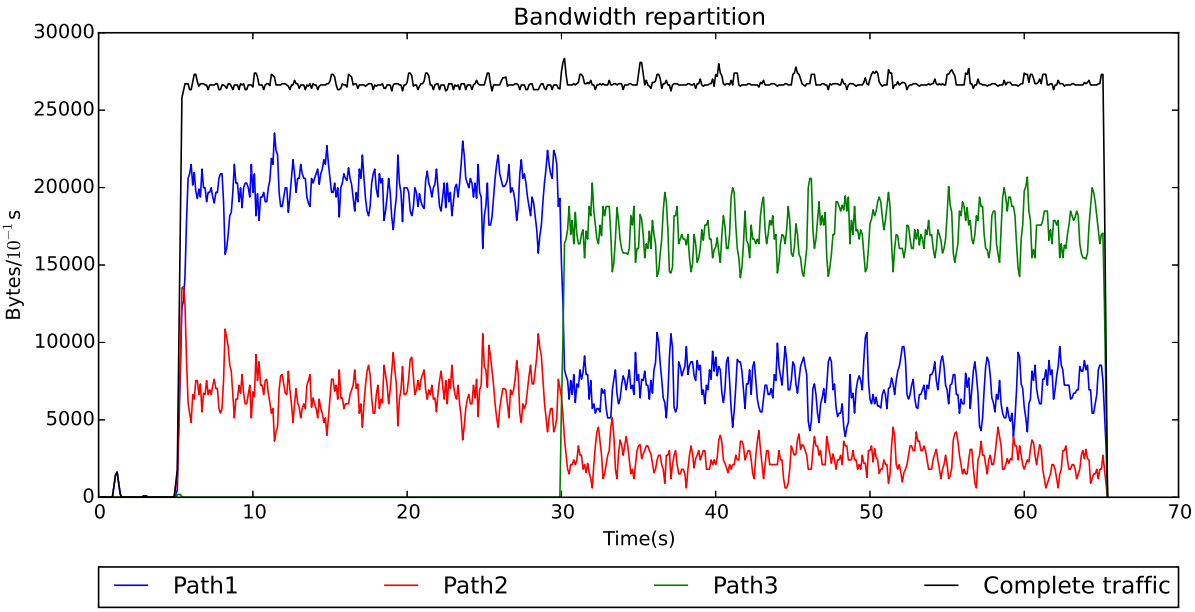

Figure 5.10: Reaction to new interface addition

The overall delay is kept reasonably small according to Figure 5.11 but we could probably optimize even more. That would imply putting more weight to path 1 and thus offloading the two other flows. However this is a choice we made to keep using all the paths while still trying to give more traffic to the faster link. It allows faster recovery if the best interface is lost.

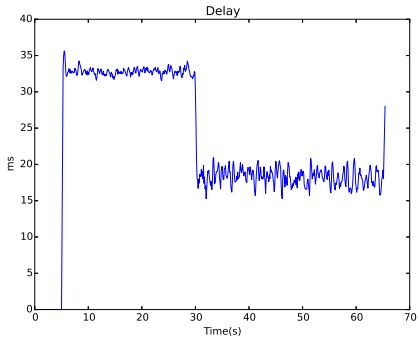

Figure 5.11: Overall delay

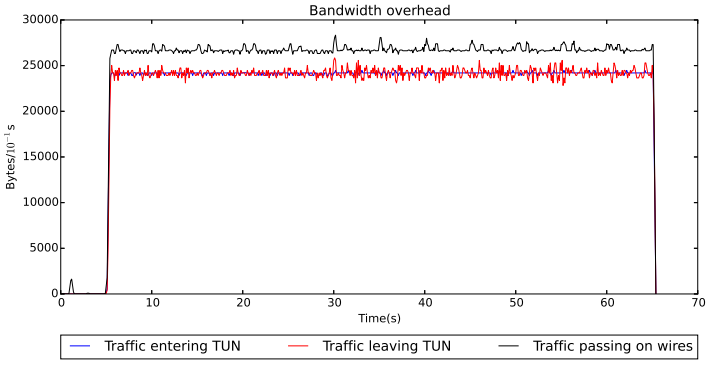

Figure 5.12: Application perception of traffic

If we look at the traffic perceived by the application on Figure 5.12, we notice no real perturbation around $30s$. Although the traffic is a bit noisy after this time, this is explained by the fact we have 3 different paths with 3 different delays and therefore packets cannot arrive at a constant rate.

5.4 Conclusion

In this chapter, we have presented a simple VPN application that we have designed to evaluate our MPDTLS implementation. To dispatch efficiently the packets between the available flows, 3 schedulers have been created : `Round Robin`, `Optimize Loss` and `Optimize Latency`. We have concentrated our evaluation on the last two since the `Round Robin` doesn't consider any contextual information.

After various experiments, we have shown that the `Optimize Latency` scheduler behaves better when there is no congestion. Otherwise, it tries to push a large amount of traffic on the faster link and quickly triggers congestion. However, we have discovered that the counter-reaction will avoid too many losses. As the additional delay caused by the congestion will increase the perceived latency, the scheduler will therefore redirect the traffic through other paths.

On the other side, the `Optimize Loss` scheduler behaves better when there is congestion or simply if one link looses more packets than another one. This would typically be the case for a Wi-Fi interface in an environment with interference. In such case, the most reliable link is preferred as long as it is not congested.

In addition, we have shown that an application using our MPDTLS tunnel will not be much troubled if one interface is lost in the middle of the communication. This is also the case if we add a new interface on the fly.

6 | Conclusion

6.1 General conclusion

In this master thesis, we have introduced a new protocol called Multipath DTLS. First we reviewed the current version of DTLS together with the principles it inherits from TLS. In addition, we presented the different messages involved in a communication as well as a typical handshake. This was important to understand how we could integrate a new extension in this existing design.

Secondly, we have presented an overview of MPRTP which is an existing multipath protocol. In this chapter, we disclosed the essential components that allow the use of multiple interfaces concurrently. Since RTP uses most of the time UDP as its transport protocol, we were able to integrate and adapt these principles into MPDTLS.

Chapter 3 describes our design together with the new messages introduced. We had to solve multiple challenges:

1. Securely exchange available addresses between hosts
2. Introduce a way to evaluate the quality of a particular path
3. Design a light handshake to establish new flows.

The first point is achieved with the help of a new message called the `ChangeInterfaceMessage` which carries all available IP addresses and ports of one host.

The second point is handled with the addition of a feedback mechanism. This gives to the sender various information about the link such as the forward delay or the loss rate perceived. The scheduler may later use this information to choose the volume of traffic it wants to send on each flow.

Finally we dealed with the last point by introducing new messages. Namely `WantConnect` and `WantConnectAck` that serve to establish a new flow if there is at least one other flow alive. All these messages have been secured as normal `Application Data` with the keys negotiated during the handshake.

Chapter 4 gave some details about the concrete implementation of this protocol. We have decided to work with wolfSSL library and we presented how the calls are handled internally. We also have reviewed our choices of structures to handle the different mechanisms needed for MPDTLS.

In the last chapter, we have evaluated our solution by building a simple VPN application which uses our modified library. We then measured the distribution of traffic between the different available paths with 2 different schedulers in various environments. We have shown that our implementation can take advantage of multiple interfaces and support more bandwidth that what would be available with only one link. Moreover, the connection is not much troubled if we loose or add an interface in the middle of the communication.

In conclusion, the initial goal of our extension (i.e. providing resiliency, mobility and better performances to DTLS when multiple interfaces are available) is fully achieved.

6.2 Future work

Our implementation of MPDTLS is functional, but could be more flexible. We have noticed some limitations when we were trying to integrate it with an existing application (see Appendix A). A future work would be to decouple the pure standard aspect such as sending or receiving MPDTLS packets from the I/O. To be more flexible we need to give the control back to the application when we want to open a new socket. This would require some engineering because we used the sockets to reserve some port numbers in advance and communicate them to the other host. A good compromise would be to stay with an internal system but making it easily replaceable by the application as we did for the scheduler. This has the advantage to bring no additional setup for a simple application but gives a way for applications with more complex requirements to customize I/O as they wish.

As we have seen in Chapter 5, we are using 2 different scheduling policies that have their best behaviour in different situations. We could investigate a way to merge them into a unique scheduler. It will compute the fractional distribution on both criteria : loss rate and forward delay. We can imagine such a scheduler giving priority to the loss rate in case of big difference between the subflows; and choosing the fastest link if the loss rates are equivalent. It would need much more experiments and efforts to design this scheduler but it seems to be an interesting thought for future works.

Another point we have put aside during the development is the NAT traversal feature. Such a functionality would be needed to use the VPN application at home. We have to investigate how to integrate STUN [34] correctly and securely to obtain IP addresses and ports of NATed interfaces. This would also imply adding more information in the data structures inside the library because the public IP and the local IP of the created socket have to be stored for each subflow.

A last point to discuss would be whether or not we should allow for DNS name inside our `Change InterfaceMessage` as MPRTP does (see Section 2.2.3). The security of DNS resolution could be guaranteed with DNSSEC [48] and the usage of this protocol would therefore become a necessity. The support for DNS could bring substantial advantages because one DNS name can reference multiple IP addresses. In particular, one DNS name may have both A and AAAA fields and therefore the application will retrieve the corresponding IPv4 and IPv6 addresses to create two separate flows.

Bibliography

[1] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176, 7465.

[2] Nagendra Modadugu and Eric Rescorla. The design and implementation of datagram tls. In *NDSS*, 2004.

[3] Christian Huitema, Eric Rescorla, and Jana Iyengar. Dtls as subtransport protocol. Internet-Draft draft-huitema-tls-dtls-as-subtransport-00, IETF Secretariat, March 2015. `http://www.ietf.org/internet-drafts/draft-huitema-tls-dtls-as-subtransport-00.txt`.

[4] Cisco. Cisco AnyConnect VPN Client. `http://www.cisco.com/c/en/us/support/security/anyconnect-vpn-client/tsd-products-support-series-home.html`, 2011.

[5] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. Tcp extensions for multipath operation with multiple addresses. RFC 6824, RFC Editor, January 2013. `http://www.rfc-editor.org/rfc/rfc6824.txt`.

[6] Varun Singh, Teemu Karkkainen, Joerg Ott, Saba Ahsan, and Lars Eggert. Multipath rtp (mprtp). Internet-Draft draft-singh-avtcore-mprtp-10, IETF Secretariat, November 2014. `http://www.ietf.org/internet-drafts/draft-singh-avtcore-mprtp-10.txt`.

[7] Jon Viega, Pravir Chandra, and Matt Messier. *Network Security with Openssl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2002.

[8] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), January 2012. `http://www.ietf.org/rfc/rfc6347.txt`.

[9] David Woodhouse et Al. OpenConnect VPN. `http://www.infradead.org/openconnect/`, 2015. [Online; accessed 17-May-2015].

[10] Whitfield Diffie, Paul C Van Oorschot, and Michael J Wiener. Authentication and authenticated key exchanges. *Designs, Codes and cryptography*, 2(2):107–125, 1992.

[11] Whitfield Diffie and Martin E Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[12] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and SheuelingChang Shantz. Comparing elliptic curve cryptography and rsa on 8-bit cpus. In Marc Joye and Jean-Jacques Quisquater,

editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer Berlin Heidelberg, 2004.

[13] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. draft-ietf-tls-tls13-05, March 2015. http://tools.ietf.org/id/draft-ietf-tls-tls13-05.txt.

[14] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International Journal of Information Security*, 1(1):36–63, 2001.

[15] Cloudfare. TLS handshake with ECDHE. https://blog.cloudflare.com/keyless-ssl-the-nitty-gritty-technical-details/, 2014. [Online; accessed 02-May-2015].

[16] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology—ASIACRYPT 2000*, pages 531–545. Springer, 2000.

[17] P. Gutmann. Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7366 (Proposed Standard), September 2014.

[18] NSFOCUS. DNS Amplification Attacks on the Rise. http://www.nsfocus.com/2013/SecurityView_0927/147.html, 2013. [Online; accessed 02-May-2015].

[19] Adam Bergkvist, Daniel C Burnett, Cullen Jennings, and Anant Narayanan. Webrtc 1.0: Real-time communication between browsers. *Working draft, W3C*, 91, 2012.

[20] Eric Rescorla. Webrtc security architecture. Internet-Draft draft-ietf-rtcweb-security-arch-11, IETF Secretariat, March 2015. http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-security-arch-11.txt.

[21] Eric Rescorla. Proposed WebRTC security architecture. https://www.ietf.org/proceedings/82/slides/rtcweb-13.pdf, 2011. [Online; accessed 28-April-2015].

[22] INFOSEC Institute. SSL Renegotiation Attack. http://resources.infosecinstitute.com/ssl-attacks/, 2013. [Online; accessed 03-May-2015].

[23] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard), February 2010. http://www.ietf.org/rfc/rfc5746.txt.

[24] Eric Rescorla. SSL/TLS and Computational DoS . http://www.educatedguesswork.org/2011/10/ssltls_and_computational_dos.html, 2011. [Online; accessed 03-May-2015].

[25] R. Seggelmann, M. Tuexen, and M. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520 (Proposed Standard), February 2012. http://www.ietf.org/rfc/rfc6520.txt.

[26] Varun Singh, Saba Ahsan, and Jörg Ott. Mprtp: multipath considerations for real-time media. In *Proceedings of the 4th ACM Multimedia Systems Conference*, pages 190–201. ACM, 2013.

[27] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (INTERNET STANDARD), July 2003. Updated by RFCs 5506, 5761, 6051, 6222, 7022, 7160, 7164.

[28] Microsoft Technet. Forefront TMG is SIP-aware. http://blogs.technet.com/b/isablog/archive/2009/10/21/forefront-tmg-is-sip-aware.aspx, 2009. [Online; accessed 19-April-2015].

[29] Javvin Technologies. *Network Protocols Handbook*. Javvin Technologies, 2005. http://books.google.be/books?id=D_GrQa2ZcLwC.

[30] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *NSDI*, volume 11, pages 8–8, 2011.

[31] Cristel Pelsser, Luca Cittadini, Stefano Vissicchio, and Randy Bush. From paris to tokyo: On the suitability of ping to measure latency. In *IMC*, 2013.

[32] Colin Perkins and Varun Singh. Multimedia congestion control: Circuit breakers for unicast rtp sessions. Internet-Draft draft-ietf-avtcore-rtp-circuit-breakers-10, IETF Secretariat, March 2015. http://www.ietf.org/internet-drafts/draft-ietf-avtcore-rtp-circuit-breakers-10.txt.

[33] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. RFC 4291 (Draft Standard), February 2006. Updated by RFCs 5952, 6052, 7136, 7346, 7371 http://www.ietf.org/rfc/rfc4291.txt.

[34] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session traversal utilities for nat (stun). RFC 5389, RFC Editor, October 2008. http://www.rfc-editor.org/rfc/rfc5389.txt.

[35] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. Sip: Session initiation protocol. RFC 3261, RFC Editor, June 2002. http://www.rfc-editor.org/rfc/rfc3261.txt.

[36] Fei Song, Hongke Zhang, Sidong Zhang, Fernando Ramos, and Jon Crowcroft. An estimator of forward and backward delay for multipath transport. *University of Cambridge, Computer Laboratory, Technical Report*, (UCAM-CL-TR-747), 2009.

[37] Quentin Devos and Loïc Fortemps. Multipath DTLS implementation inside wolfSSL. http://dx.doi.org/10.5281/zenodo.17919, 2015.

[38] Wikipedia. Datagram transport layer security. https://en.wikipedia.org/wiki/Datagram_Transport_Layer_Security#Implementations, 2015. [Online; accessed 20-May-2015].

[39] wolfSSL Inc. wolfSSL Embedded SSL Library. https://github.com/wolfSSL, 2015. [Repository; latest release in 2015].

[40] wolfSSL Inc. wolfSSL Embedded SSL Library. https://yassl.com/yaSSL/Home.html, 2015. [Online; accessed 20-May-2015].

[41] Laura Chappell. *Wireshark 101: Essential Skills for Network Analysis*. Protocol Analysis Institute, Chapell University, 2013.

[42] wolfSSL Inc. wolfSSL Manual. http://www.yassl.com/yaSSL/Docs-cyassl-manual-toc.html, 2015. [Online; accessed 24-May-2015].

[43] Florent Bondoux. Campagnol : distributed vpn over udp/dtls. http://campagnol.sourceforge.net/, 2011.

[44] Loïc Fortemps and Quentin Devos. VPN Application using Multipath DTLS. http://dx.doi.org/10.5281/zenodo.17920, 2015.

[45] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[46] Benjamin Hesmans. Simple tool, based on mininet, to boot a simple network with n paths and run experiments between two hosts. https://bitbucket.org/bhesmans/minitopo, 2015.

[47] Alessio Botta, Alberto Dainotti, and Antonio Pescapè. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks*, 56(15):3531–3547, 2012.

[48] S. Weiler and D. Blacka. Clarifications and implementation notes for dns security (dnssec). RFC 6840, RFC Editor, February 2013. http://www.rfc-editor.org/rfc/rfc6840.txt.

[49] Inc. Docker. Build, Ship and Run Any App, Anywhere. https://www.docker.com, 2015. [Online; accessed 27-May-2015].

[50] Quentin Devos and Loïc Fortemps. Graph Generation Scripts. https://gist.github.com/Okhoshi/dbc550b3133aab4dc505, 2015. [Online; accessed 29-May-2015].

Glossary

CDN Content Delivery Network

CIM Change Interface Message

DH Diffie-Hellman

DoS Denial of Service

DTLS Datagram Transport Layer Security

ECDHE Elliptic Curve Diffie-Hellman Ephemeral

EWMA Exponential Window Moving Average

Flow is a connection between two hosts. A flow may be composed of multiple subflows to take advantage of multiple interfaces.

HMAC keyed-Hash Message Authentication Code

HTTPS Hypertext Transfer Protocol over TLS

IP Internet Protocol

MPRTP Multipath Real-time Transport Protocol

MPTCP Multipath TCP

NAT Network Address Translation

RTCP RTP Control Protocol

RTP Real-time Transport Protocol

RTT Round-trip Time

SIP Session Initiation Protocol

SRTP Secure Real-time Transport Protocol

Subflow is a connection between two interfaces, each of one possessing one IP address and one port number.

TBD To Be Defined

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

VoIP Voice over IP

VPN Virtual Private Network

Appendices

A | Campagnol VPN

In this appendix, we detail the architecture of Campagnol [43] and explain why we couldn't make the integration with wolfSSL [40].

A.1 Campagnol architecture

First of all, Campagnol is a decentralized VPN in the sense that there is no server. Every user runs a peer and the peers communicate together directly. Each peer has to define its own IP address inside the VPN (i.e. 10.0.0.1 for peer 1, 10.0.0.2 for peer2, etc.). The first phase for a peer is to register with the rendez-vous server. Figure A.1 is depicting this phase. Obviously, two peers cannot claim the same VPN IP.

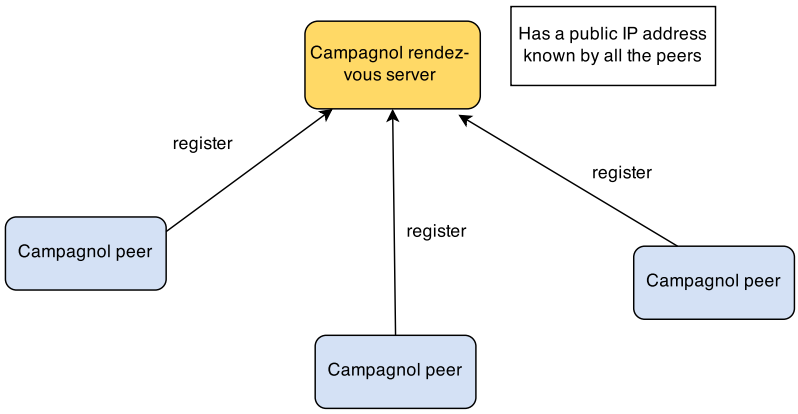

Figure A.1: Every peer has to register with the rendez-vous

Whenever one of the peer wants to access another VPN IP, it contacts the rendez-vous server and asks for the real IP if it exists. Figure A.2 presents these interactions. Once the IP address and the port of the corresponding peer have been retrieved, the DTLS handshake takes place.

If we go deeper in the implementation, we notice that all the code is threaded and each thread has its own responsibility. For instance all the communications with the rendez-vous server and the other peers are using the same and only UDP socket. The thread handling the socket is then responsible to dispatch the packets to the other threads through a FIFO queue. Figure A.3 is depicting this behavior.

In order to know if a particular packet is DTLS, the record layer is extracted and the sender's IP address is matched against the list of known peers. It is then transmitted to the thread in charge of this session. The replies, if replies exist, are transmitted using again the same UDP socket.

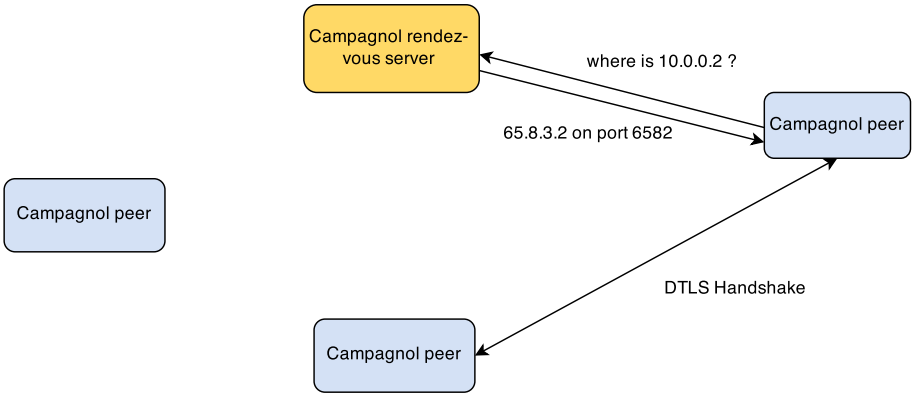

Figure A.2: Real IP acquisition through RDV server

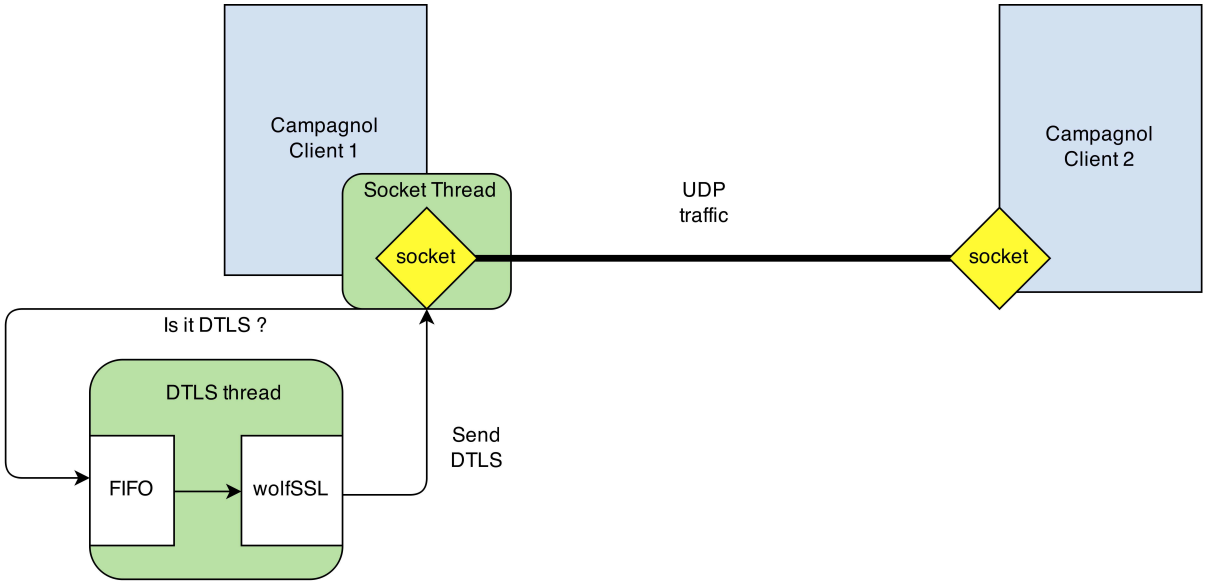

Figure A.3: Campagnol DTLS communications

A.2 wolfSSL integration difficulties

We faced some difficulties trying to integrate successfully our modified library before we realized it was really difficult to both keep the decentralized aspect and bringing multipath. This would have required to redo the whole design of Campagnol and this was not our objective.

Here is a list of the main difficulties :

- The socket control : we modify wolfSSL and integrate the handling of the sockets when flows are created or removed. It was a design choice but it brings some limitations on the existing application that could use this library. We need a full control of the sockets and the ability to create new ones that have the same status of the original one (used for the handshake). This is clearly not possible here because as shown in Figure A.3, all the traffic is using the same non-connected UDP socket.
- the CBIO compatibility : Campagnol originally uses OpenSSL [7] with the CBIO. This is an I/O

abstraction provided by OpenSSL which can be useful but is only partially supported by wolfSSL. The whole system of FIFO queues between threads is using this CBIO abstraction and we had to rethink about a system that could be compatible with wolfSSL.

- The nature of sockets : Campagnol uses non-blocking, non-connected sockets. While our implementation is compatible with non-blocking sockets as we tried not to brake this feature inside the library, we absolutely need connected sockets (as wolfssl do) to work correctly. When we choose a flow to send a particular packet, we only give the socket number to the `sendto` method. Even if we could proceed differently, we have chosen this way because wolfSSL requires this in the first place.

For all these reasons, we decided to develop our own VPN application between a client and a server like OpenVPN. Campagnol was still useful because we had to understand how it works quite deeply and we were able to take some parts of the code for our application; namely, the TUN initialization and the bridge between the TUN and the SSL library.

B | Summary of the MPDTLS syntax

This section aims to do a summary of all the changes we made to the DTLS 1.2 RFC [8]. All the added fields are explained in the Chapter 3, the DTLS fields are explained in the RFC6347 and finally the TLS fields are detailed in the RFC5246 [1]. For the sake of clarity, the TLS fields are the uncommented fields.

B.1 Record Layer

```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 epoch;                                          // DTLS field
    uint48 sequence_number;                                // DTLS field
    uint16 length;
    select (ContentType) {
        case application_data: opaque fragment[DTLSPlaintext.length];
        case change_interface: ChangeInterfaceMessageFragment;  // MPDTLS addition
        case want_connect: WantConnectFragment             // MPDTLS addition
        case want_connect_ack: WantConnectAckFragment      // MPDTLS addition
        case Feedback: FeedbackFragment                    // MPDTLS addition
        case Feedback_ack: FeedbackAckFragment             // MPDTLS addition
    } fragment;
} DTLSPlaintext;

enum {
    change_cipher_spec(20),
    alert(21),
    handshake(22),
    application_data(23),
    change_interface(TBD),                                 // MPTDLS addition
    want_connect(TBD),                                     // MPTDLS addition
    want_connect_ack(TBD),                                 // MPDTLS addition
    feedback(TBD),                                         // MPTDLS addition
    feedback_ack(TBD),                                     // MPDTLS addition
    (255)
} ContentType;


struct {                                                   //
    byte reply;                                            //
    if (reply) {                                           //
        uint48 ack;                                        //
    }                                                      //
```

```
    byte number_addresses;                                  //
    NewAddress addresses<1..2^8-1>;                         //
} ChangeInterfaceMessageFragment;                           // MPDTLS addition

struct {
    NewAddress addr_src;                                    //
    NewAddress addr_dst;                                    //
    byte opts;                                              //
} WantConnectFragment;                                      // MPDTLS addition

struct {                                                    //
    uint48 ack_seq;                                         //
    byte opts;                                              //
} WantConnectAckFragment;                                   // MPDTLS addition

struct {                                                    //
  uint64 received_packets_count;                            //
  uint48 min_sequence_number;                               //
  uint48 max_sequence_number;                               //
  uint64 average_forward_delay;                             //
} FeedbackFragment;                                         // MPDTLS addition

struct {                                                    //
  uint32 feedback_sequence_number;                          //
} FeedbackAckFragment;                                      // MPDTLS addition

struct {                                                    //
    byte address[16];                                       //
    uint16 port;                                            //
} NewAddress;                                               // MPDTLS addition
```

B.2 TLS Hello Extension

As shown in Section 3.1.1, we use TLS Hello Extensions [13](from Section 7.3.2.5) with a new `Extension Type` to advertise the MPDTLS ability of a host.

```
struct {
    ExtensionType extension_type;
    byte          extension_length[2];
    byte          extension_data[1];
} Extension;

enum {
    mpdtls_extension(TBD), (65535)                          // MPDTLS addition
} ExtensionType;
```

C | Instructions to generate graphs

This appendix aims to detail the steps followed to get the figures presented in the Chapter 5.

C.1 Installing tools

C.1.1 Manual install

If you do not want to use Docker, the manual installation steps are presented in the following sections. Otherwise, you should go directly to Section C.1.2 to proceed with the Docker image.

Installing the library

The first step is of course to install our modified version of wolfSSL [37]. Be careful to look at the ./configure -h option which describes all the possible features you may want to enable. For mpdtls to work, you should at least use the flag `-enable-mpdtls`. Other options may be needed depending on which cipher suite you will use for the handshake. Since wolfSSL targets embedded devices, most of the non-essential features are turned off by default. This is the case for the elliptic curve cryptography and the Diffie-Hellman key exchange.

Most of the time during the writing of this thesis, wolfSSL has been configured with the following options:

- `-enable-mpdtls`
- `-enable-debug`
- `-enable-dh`
- `-enable-ecc`
- `-disable-oldtls`

Installing Mininet

Mininet[45] is a useful tool to emulate a specific network topology on a simple computer. Depending on your Linux distribution, you will need to add other packages to make it work. Our experiments are taking

place inside a Mininet environment but with the help of Minitopo[46]. This is a Python framework to easily configure the topology and run experiments based on little configuration files.

Instructions to install Mininet can be found on its website [45].

Installing D-ITG

D-ITG [47] is the Internet traffic generator we use to simulate traffic on top of our MPDTLS tunnel. On most distributions, it can be installed via apt-get.

```
sudo apt-get install d-itg
```

Installing Minitopo

At the time of writing, the Minitopo repository[46] is not yet publicly available, but it should be the case soon. Once the repository is accessible, simply download it and put the content of the `src` folder into the experimentation directory. Another possibility is to put that folder at any place on the computer and add it to the `$PYTHONPATH` environment variable.

The only requirement is that the `mpPerf.py` must be in the experimentation folder.

Setting up the VPN application

Our MPDTLS VPN application [44] must simply be downloaded and compiled with `make`. We have modified Minitopo to run the `Client` and `Server` executables on different Mininet entities. These executables must be either compiled or copied after compilation inside the experimentation folder.

C.1.2 The easy way

We have built a Docker [49] image providing the whole test environment. Once Docker is installed (installation guides are provided for each supported operating systems), the image of our environment can be launched with:

```
docker run --privileged multipathdtls/mpdtls-testbed:latest
```

If the image does not exist locally, Docker will take care of downloading it from Docker Hub. As mininet performs some modifications in the interfaces configuration, the Docker container needs absolutely to be ran with the `-privileged` flag. When the container is ran without any additional parameter, minitopo is launched with the default scripts we provide in the image.

C.2 Experiments

Now that the testing environment is set up, we can start running some experiments within the lab. The base topology of the lab is fixed and was presented in the Section 5.3.1. However, the number of paths between Client and Router can be configured, as well as their specifications.

C.2.1 Define the configuration

The configuration of the experiment is based on at least two configuration files: `conf` and `xp`.

conf

The `conf` file controls the topology of the lab, as the number of paths between the Client and the Router for instance. A typical `conf` file is presented in the Listing C.1. A line starting with # is a comment and not considered to build the topology.

```
topoType:MultiIf
leftSubnet:11.0.
rightSubnet:11.2.
# path_x:delay,queueSize(may be calc),bw(Mb),loss
path_0:30,1000,5,0
path_1:40,1000,5,0
path_2:10,1000,5,0
path_3:0,1000,10,0
```

Listing C.1: Typical configuration file

The first line defines the type of Topology that will be used. MULTIIF is the only type that interests us.

`leftSubnet` and `rightSubnet` defines the IP prefixes that will be used between the Client and the Router and between the Router and the Server respectively.

The last lines, beginning with `path_`, defines the paths that must be created between the Client and the Router. The first number is the delay of that path, expressed in milliseconds. The second is the queue size of the interface. As we do not vary this parameter, we let it to its maximal value, 1000. The third is the bandwidth of the link, in Mbps. The last one is the loss rate, in percent.

For instance, the `conf` presented in the Listing C.1 would produce the same topology as presented in the Figure 5.3.

xp

The `xp` file tells Minitopo which experiment to run and how to run it. Several experiments are implemented in Minitopo, but we will focus on the `itg` one, because it involves D-ITG. The other types are explained in the Minitopo documentation[46].

```
xpType:itg
vpnSubnet:20.0.
scheduler:2 100
script:script.itg
```

Listing C.2: Typical experience file

As said before, we focus on the ITG experiment, and this is materialized by the first line in Listing C.2. The second line, `vpnSubnet`, defines the subnet used by our VPN application for the TUN interfaces. It must be different than the subnet used for minitopo in the `conf` file.

The `scheduler` option defines which scheduler to use during the experiment. The value is directly transmitted to the VPN client and so must follow the format `N°Scheduler[space]Granularity`. The available schedulers are :

1. RoundRobin
2. OptimizeDelay
3. OptimizeLoss

The last option is mandatory when running an ITG experiment. It defines the script to give to ITGSend (which is ran in Multiflow mode). The structure of this file is explained in the next section.

The `xp` is mandatory but might be empty. In this case, the Mininet cli will appear instead of launching an experience.

ITG script

This configuration file, called `script.itg` by default, defines the flow that must be started by D-ITG. Each line is a flow to start, and can contain every option of ITGSend. These options are explained in the documentation of D-ITG[47].

```
-a 20.0.0.1 -rp 8900 -T UDP -t 30000 -c 781.25 -C 300
```

Listing C.3: Default content of `script.itg`

C.2.2 Run the lab

The lab is simply ran by either

```
sudo ./mpPerf.py -t conf -x xp
```

if all the lab components were installed manually, or

```
docker run --privileged -ti multipathdtls/mpdtls-testbed:latest
```

if the docker way was preferred.

The results of the experiment are available directly in the directory were `mpPerf.py` was launched. However, if docker was used, the results are stored in a Docker Volume. This volume can be synchronized with a selected folder with the option `-v /path/to/folder:/experience/data` added before `multipathdtls/mpdtls-testbed`. `/path/to/folder` must be the absolute path towards the folder.

Also, the configuration files may be modified by using the following options:

```
-v /path/to/conf:/experience/conf
-v /path/to/xp:/experience/xp
-v /path/to/script.itg:/experience/script.itg
```

Complete experimentation

To help us in the generation of the Figures 5.5 and 5.6, we wrote two scripts that run 11 times each configuration, with 48 possible configurations (TCP or UDP, Scheduler `OptimizeLoss` or `OptimizeDelay`, 4 links characteristics variations and 3 possible total generated bandwidths). These scripts are available on our Gist [50].

These scripts are already embedded in the Docker image, and can be run by simply adding either `bandwidth` or `losses` at the end of the `docker run` command.

C.2.3 Generate graphs

Once the experiments are done, multiples files are created:

- Log files (client, server, lab, command). These files contain the output of the various components of the lab. The most interesting one is `lab.log`, generated only if the experiments were launched with our scripts. Otherwise, its content is simply the output of the Minitopo execution.
- Err files (client and server). These files contain the debug output of the client and server of our VPN application.
- Pcap files. These files are the most important as they are the foundations to draw the graphs. They need a working installation of tcpdump to be generated.

In the Gist [50], there are three python scripts that plot graphs using Matplotlib. In order to use these scripts, some requirements must be followed.

These python packages must be installed:

- matplotlib

- numpy
- scipy

Tshark is also necessary to generate the csv files from the pcap files. Tshark should be installed with Wireshark.

In general, the python scripts do not need to be called directly. For instance, the Figure 5.5 can be plotted right after the run of `bandwidth_graph.sh` simply by rerunning the script but with the `draw` argument.

In the case of isolated experiments, some graphs can be plotted with the `plot.sh` script, also available in the Gist. This script needs that the files generated during the experiments are put in a `data` folder.

D | Code contribution

D.1 WolfSSL

The table below is a `git diff` between our wolfSSL repository and the official one. It gives an idea about the work done to implement our MPDTLS extension. This can be reproduced dynamically on our Github repository page [37].

.gitignore	+1 -1
README	+7 -0
README.md	+6 -0
configure.ac	+36 -0
src/internal.c	+1,844 -65
src/io.c	+77 -3
src/ssl.c	+358 -3
src/tls.c	+187 -2
wolfssl/error-ssl.h	+5 -2
wolfssl/internal.h	+253 -4
wolfssl/ssl.h	+59 -1
wolfssl/wolfcrypt/error-crypt.h	+5 -0
wolfssl/wolfcrypt/types.h	+3 -1

The largest part of our work is concentrated inside the `internal.c` file. This is really the core of the library since it treats every messages. It is in this file that all the *Do*Functions and the *Send*Functions described in Chapter 4 are implemented. All the initialization and creation of the structures used to store information about the flow including feedback and heartbeat are also done there.

All the API calls are implemented inside `ssl.c` such as the `use_multipath()` or the `add_new_addr` functions. They are checking the input data from the user and make the appropriate calls inside `internal.c`.

The heartbeat extension has required some modifications inside `tls.c` to be fully supported. This is also where we had to put our Hello extension for the extension discovery during the handshake.

The file `io.c` has slightly been modified to integrate our schedulers outside the core of the library and make it easily replaceable.

Some minor modifications have also be made in other files to support our new option `-enable-mpdtls` during the building process.

D.2 Wireshark

In this section we present our contribution to Wireshark [41] in order to support our new messages. As you can see we have mainly modified the dtls dissector to detect our packets and indicate how to format them once they are decrypted.

.gitignore	+2 -0
epan/dissectors/packet-dtls.c	+444 -0
epan/dissectors/packet-ssl-utils.c	+11 -0
epan/dissectors/packet-ssl-utils.h	+7 -1
epan/dissectors/packet-ssl.c	+5 -0

Figure D.1 is a typical MPDTLS communication we can observe with Wireshark. Note that all the new packets are correctly identified.

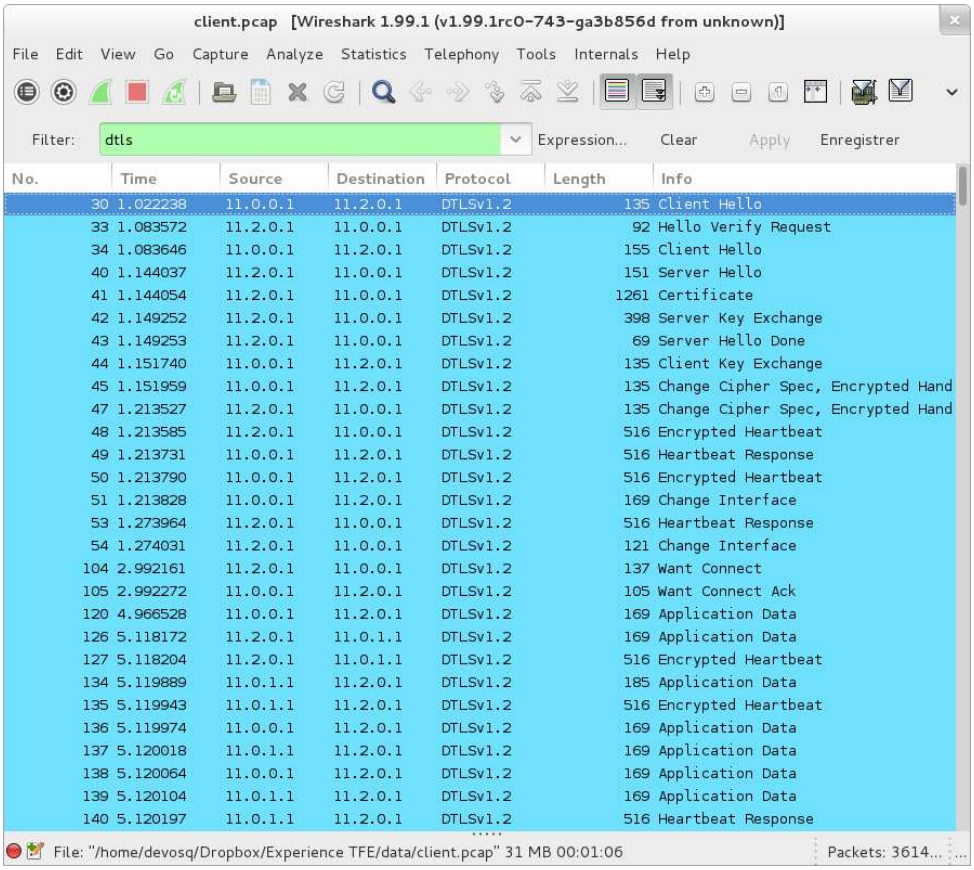

Figure D.1: Wireshark trace of MPDTLS traffic

An example of how a `CIM` packet is formatted with Wireshark is shown on Figure D.2. We can see all advertised addresses together with their port number.

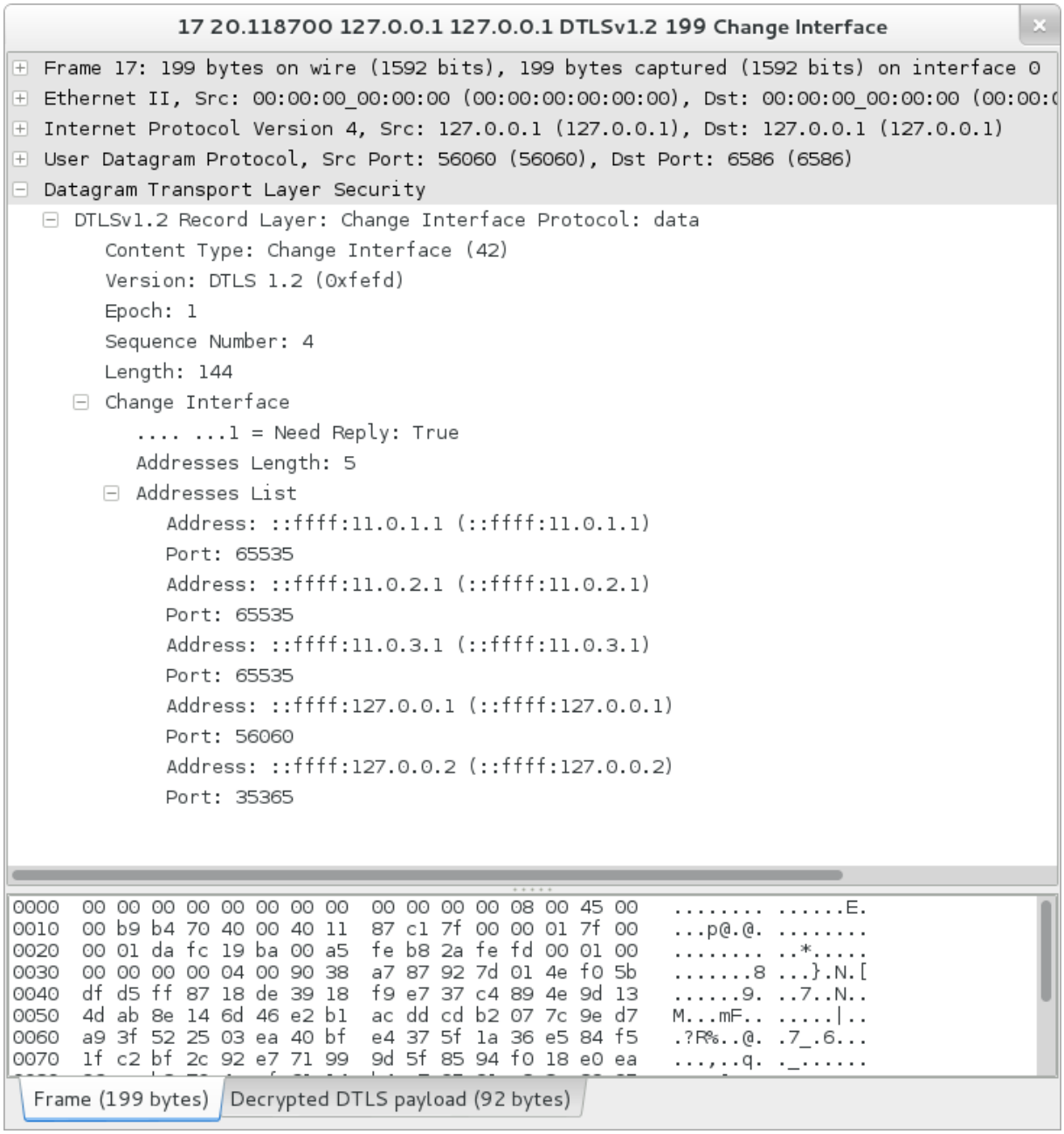

Figure D.2: The decrypted content of a CIM with Wireshark