ALMA MATER STUDIORUM
UNIVERSITY OF BOLOGNA

Department of Computer Science and Engineering

Master's Degree in Computer Engineering

MASTER'S THESIS

in

Protocols and Architectures for Space Networks M

Design and implementation of a DTN Convergence Layer Adapter based on the QUIC protocol

Candidate

Mattia Moffa

Supervisor

Prof. **Carlo Caini**

Co-supervisor

Dott. Ing. **Tomaso de Cola**

Academic Year 2023/2024

Abstract

TCP/IP, the protocol suite powering the Internet, owes its name to its two foundational protocols: IP, which ensures each packet is routed over the network to its destination, and TCP, which deals with end-to-end recovery, reordering, flow and congestion control.

The TCP/IP architecture, however, makes a few assumptions which do not apply to all networks. In particular, it presumes an end-to-end communication link with minimal disruption, a sufficiently low error rate and small, consistent latency. Networks that cannot satisfy these assumptions, and thus cannot support the TCP/IP architecture, are called *challenged networks*. The first studied case of a network that exhibits these characteristics is that of *InterPlanetary Networks* (IPN); however, similar conditions were later found in other scenarios, such as sensor, satellite, underwater and military tactical networks, as well as any sufficiently isolated environments, such as emergency networks.

The DTN architecture (*Delay/Disruption-Tolerant Networking*) derives from an extension of the scope of Interplanetary Networking to all challenged networks. Its objective is exactly that of relaxing the implicit requirements of the TCP/IP architecture in order to guarantee efficient communication in such networks.

The DTN architecture is described in RFC 4838, which introduces a new protocol layer, the Bundle Layer, located between Application and Transport. The protocol that operates at this layer is the Bundle Protocol, described in detail in IRTF RFC 5050 (version 6) and recently standardized by IETF in RFC 9171 (version 7); its role is to enable applications to transmit self-contained messages (*bundles*) over a challenged network. While Internet protocols assume that when a router receives a packet it is able to immediately send it to the next node, the Bundle Protocol relies on a store-and-forward mechanism, where each bundle is stored locally (normally in persistent memory) until the link to the next node becomes available.

The protocol used by a DTN node to transfer a bundle to the next DTN node (forming a single DTN hop) is independent of those used on other hops. For example, DTN hops that traverse Internet-like networks can use TCP, while DTN hops in deep space must use the LTP protocol, designed exactly for this purpose. The protocol stack used to send a bundle is called *Con-*

vergence Layer in its entirety. The protocol concerned with sending bundles over a specific convergence layer is called *Convergence Layer Adapter* (CLA). The TCP Convergence Layer Adapter (TCPCL), which transfers bundles over a TCP connection, was described in 2014 in RFC 7242 (version 3) and standardized in RFC 9174 (version 4).

In 2012, TCP's perceived limitations prompted Google to design and develop a new transport protocol called QUIC, meant to replace TCP in HTTP communications. Although the project was initially controlled by Google, a QUIC working group was established in 2016, and the protocol was later standardized by IETF in 2021, with the RFCs 8999–2002.

The objective of this thesis is to design QUICCL, a Convergence Layer Adapter that transfers bundles over a QUIC connection, and develop an implementation for the Unibo-BP implementation of the Bundle Protocol. The QUICCL specification is partly based on that of TCPCLv4, with due modifications, while the implementation makes use of Picoquic, a QUIC implementation widely used in research.

This document also includes insights into the development of a Wireshark dissector for the newly designed protocol to support its functional evaluation.

This work has been carried out at the Institute of Communication and Navigation of the German Aerospace Center (Deutsches Zentrum für Luft- und Raumfahrt, DLR) located in Oberpfaffenhofen, Munich, in cooperation with the University of Bologna, in parallel with two companion theses by fellow students Luca Andreetti and Valentino Cavallotti. The three theses constitute a larger research project aimed at exploring the usefulness of QUIC in satellite networks and challenged networks in general.

All developed software was released as free software under the GNU GPLv3 license and is freely downloadable from Git repositories.

Contents

Chapter 1 Introduction

1.1 DTN

The Internet is a global system, conceived at the end of the 1970s, that allows a large number of heterogeneous computer networks to communicate through the use of TCP/IP, a packet-switching protocol suite named after its two most important protocols, TCP (Transmission Control Protocol) and IP (Internet Protocol).

The Internet architecture makes a number of assumptions which cannot be satisfied in certain environments, namely:

- that there is a continuous, uninterrupted path between source and destination for the entire duration of the communication;
- that retransmissions based on end-to-end feedback are an effective means for repairing errors;
- that end-to-end loss is relatively rare;
- that all routers and stations support TCP/IP;
- that applications need not worry about communication performance;
- that endpoint-based security mechanisms are sufficient for meeting most security concerns;
- that packet switching is the most appropriate abstraction for interoperability and performance;
- that selecting a single route between sender and receiver is sufficient for achieving acceptable communication performance.

Networks which do not satisfy at least one of these assumptions are called *challenged networks*. The DTN architecture defines an infrastructure which relaxes most of these constraints in order to allow communication in such networks, by adopting certain design principles:

- Variably sized, potentially long messages as a base abstraction, replacing streams (TCP) and datagrams of limited size (UDP, IP);
- A new node identification syntax that supports a wide range of naming and addressing

conventions in order to increase interoperability;

- Network nodes capable of long-term storage in order to allow communication between entities which lack a continuous end-to-end path;
- Security measures that protect the infrastructure from unauthorized use by discarding illegitimate traffic as quickly as possible;
- Greater ability for applications to control quality of service, delivery options and extension of data lifetimes, which allows the network to better fulfill their needs.

In order to follow these principles, the new *Bundle Protocol* (often shortened to BP) was introduced, which occupies a new layer in the protocol stack called the *Bundle Layer*. The messages exchanged by means of the Bundle Protocol are called *bundles*.

The Bundle Protocol was initially described in 2007 as version 6 (*bpv6*) by the Internet Research Task Force [RFC5050] and later standardized in January 2022 as version 7 (*bpv7*) by the Internet Engineering Task Force [RFC9171]. Version 6 was also standardized by the Consultative Committee for Space Data Systems (CCSDS) in a Blue Book [CCSDS-734.2-B-1].

In the same way that the Internet Protocol is tasked with routing datagrams through multiple LANs (Local Area Networks), and thus acts as an overlay on the Link and Physical layers of each specific hop, the Bundle Protocol routes bundles through multiple networks, as an overlay on the Transport, Network, Link and Physical layers that each of these networks operates on. The stack that lies beneath the Bundle Layer in its entirety is called *Convergence Layer*.

A *DTN hop* is therefore defined as the macro-hop between two DTN nodes, which might include multiple lower-level (e.g. IP) hops, depending on the nature of the Convergence Layer in use for that specific DTN hop.

As the Bundle Layer is meant to operate on top of multiple Convergence Layers, another protocol is added between the Bundle Layer and the Convergence Layer, called the *Convergence Layer Adapter* (CLA), the purpose of which is to deliver bundles to the next DTN hop by using the corresponding Convergence Layer. Each Convergence Layer is required to have a Convergence Layer Adapter implemented for it in order to be used with the Bundle Protocol.

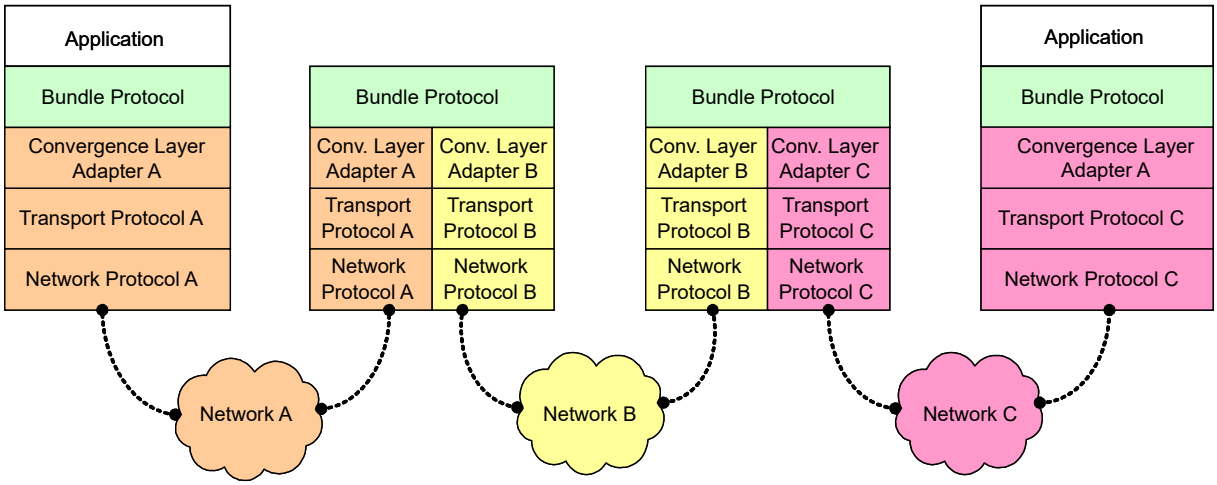

Figure 1.1: Example of operation of the Bundle Protocol over different Convergence Layers in different DTN hops.

Some existing, standardized Convergence Layer Adapters are:

- the TCPCLA, for the typical TCP/IP stack, useful on non-challenged DTN hops, such as those between two nodes on the same planet;
- the LTPCLA, for the Licklider Transmission Protocol, aimed at reliable transmission in interplanetary space.

The usefulness of this architecture lies in the fact that, since challenged networks are usually characterized by intermittent links, a *store-and-forward* approach is necessary, in which each node along the route is capable of keeping bundles in (preferably persistent) memory for long periods of time, until a link to the next node is available. For example, one may want to exchange a bundle between two nodes on two separate planets employing two relays, such as two satellites, each orbiting one of the two planets; due to the movement of the planets and the satellites, it is not always possible to establish an uninterrupted path between the endpoints. Thanks to the DTN architecture, each node can store the bundle until a link to the next node is available, instead of waiting for an end-to-end path that may not ever exist.

Nodes in the DTN network are identified by DTN *Node Identifiers* (*Endpoint Identifiers* in Bundle Protocol version 6), which follow the URI (Uniform Resource Identifier) syntax [RFC3986]. Two different formats have emerged for these identifiers:

- The ipn schema (supported by NASA's Jet Propulsion Laboratory) uses a pair of integers,

the first of which (*node number*) identifies the node the application resides on and the second of which (*service number*) identifies the application within the node (example: `ipn:6.2000`). Including a service number is mandatory: in order to indicate the node's Bundle Protocol Agent (i.e. the daemon that implements the bundle protocol), a service number of `0` is used. There is currently a proposed update to the IPN schema [Draft-IPN-update] which would introduce new features, including:

  - a null node ID to be interpreted as "nowhere" (`ipn:0.0`);
  - a node number of $2^{32} - 1$ to identify the local node, which can alternatively be shortened as `!` in the textual representation (e.g. `ipn:!.2000`);
  - an optional allocator prefix to identify the organization that assigned the node number (e.g. `ipn:12.6.2000`), which defaults to `0`, i.e. IANA, when unspecified;

- The DTN schema (supported by other entities) uses a string (*node name*) to identify the node and another string (*demux token*) to identify the application (example: `dtn://machine-26/dtnperf`); in this case, the demux token can be omitted to refer to the Bundle Protocol Agent.

1.2 Unibo-DTN and Unibo-BP

Unibo-DTN is an implementation of the DTN architecture by the University of Bologna, originally developed to facilitate DTN research without having to rely on third-party projects [Unibo-DTN]. It includes the core project Unibo-BP [Unibo-BP] (an implementation of Bundle Protocol version 7), Unibo-CGR (an implementation of Contact Graph Routing) and Unibo-LTP (an implementation of the Licklider Transport Protocol, to be used as a convergence layer).

Unibo-BP includes a TCP Convergence Layer version 3 implementation and some basic applications, such as `unibo-bp-ping`, `unibo-bp-echo`, `unibo-bp-send` and `unibo-bp-sink`. It is written in C++ 20, but it offers a C public API, and it is licensed under the GNU GPLv3 license.

The Unibo-BP ecosystem is represented in Figure 1.2.

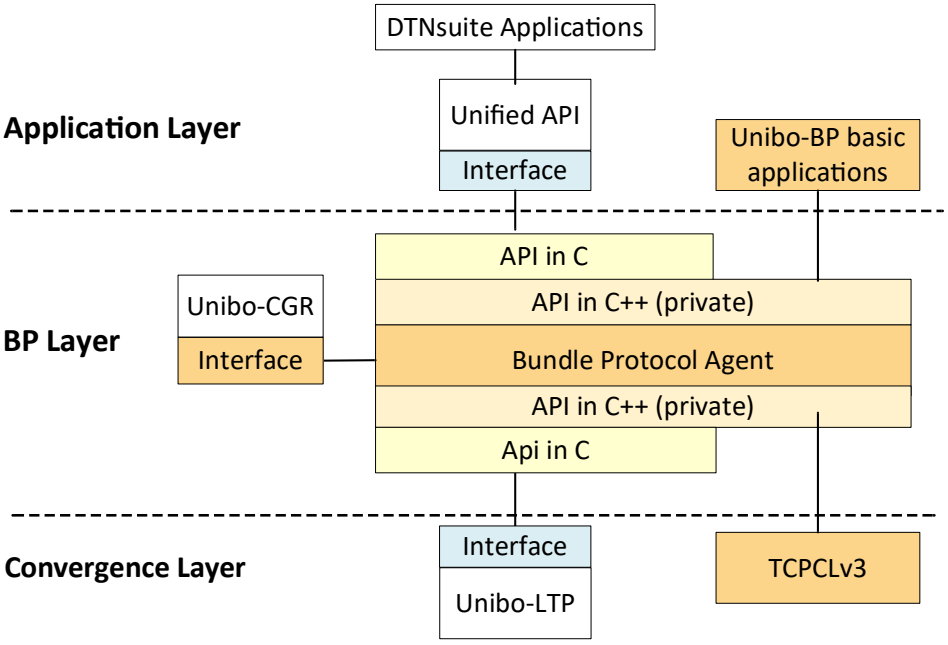

Figure 1.2: The Unibo-BP ecosystem.

Unibo-BP is structured as a set of separate processes that communicate by means of Unix sockets. This modular design enables the addition of new modules without having to alter the core Unibo-BP implementation. One example of this is Unibo-LTP, which is a completely separate project that was originally meant for ION (another implementation of the Bundle Protocol, by the NASA Jet Propulsion Laboratory).

Unibo-BP is composed by several libraries (see Figure 1.3):

- *BP* contains the core implementation of BPv7;
- *IPC* and *Client/Server* contain auxiliary features for Inter-Process Communication via Unix stream sockets;
- *CLA* contains a client/server implementation useful for the addition of new convergence layer adapters, together with the included TCPCLA implementation;
- *API* contains the public API of Unibo-BP, which is composed of C wrappers of internal C++ functions, to be used by external programs;
- *IO*, *storage*, *time* and *math* are additional utility libraries.

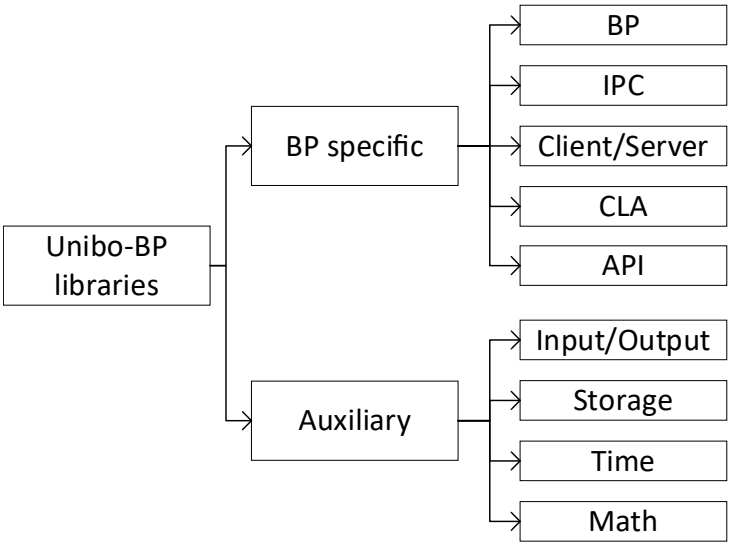

Figure 1.3: The Unibo-BP libraries.

Some peculiarities of Unibo-BP are the following:

- It supports remote administration: a client can send commands to the Bundle Protocol Agent in the form of bundles. Other popular BP implementations, such as DTNME and µD3TN, offer a similar feature through the use of TCP connections, which is more limited in that it requires the client and the server to reside in the same well-connected (non-challenged) portion of the network.
- Its local configuration is done by means of IPC commands rather than configuration files; an administration program, called `unibo-bp-admin`, is included to issue them. The result is that configuration is not done in a traditional configuration file, but rather in a shell script, which is much more powerful, and additional configuration commands can be issued at any time after the initial launch.
- It is able to simulate multiple DTN nodes on a single machine, without the need to use virtual machines; this can be particularly useful for development and evaluation purposes.
- It is supported by the Unified API, another project by the University of Bologna with the aim to provide a single API for the most important DTN implementations (including ION, DTNME, µD3TN, IBR-DTN and Unibo-BP) [UnifiedAPI]. This means that a BP application written for the Unified API can be compiled to work with any of these Bundle Protocol implementations.

1.3 QUIC

QUIC is a connection-oriented transport protocol initially introduced by Google in 2012 as a replacement for TCP with the goal of improving the performance of HTTP applications. Later, in 2021, it was standardized by IETF in [RFC9000], devoted to the protocol itself, [RFC9001], devoted to the integrated TLS security mechanisms, and [RFC9002], related to congestion control. QUIC is designed to be used on top of UDP, mainly to allow compatibility with existing network infrastructure (e.g. routers and firewalls), which may block IP datagrams that carry unrecognized transport protocol data, and to simplify the use of existing user space operating system APIs, which are better equipped to handle TCP and UDP than raw IP sockets.

From the Application Layer's point of view, the main peculiarity of QUIC compared to TCP is the availability of multiple data streams per connection. While same-order delivery is preserved within each stream, data sent in separate streams are not necessarily delivered in order. In HTTP's case, the main advantage of this feature is its effectiveness against *head-of-line blocking*, which occurs when one or more HTTP requests received out of order are forced to wait for the delivery of earlier requests before being handed over to the application. Losses may aggravate the issue even further. The only solution that can be employed with TCP is the use of multiple connections between the same two nodes, which is however not ideal due to the limited number of connections available to the browser and due to the fairness problem that would cause: multiple TCP connections would be influenced by separate congestion control parameters and thus take up the network's fair share multiple times.

Among the advantages over TCP offered by QUIC are also the following:

- Establishing a TCP connection that employs Transport Layer Security involves a typical TCP three-way handshake followed by a TLS handshake, resulting in a delay of three round-trip times before any application data can be sent. QUIC, on the other hand, embeds the TLS handshake in the transport handshake, allowing data to be sent after as little as one round-trip. In addition, QUIC allows the applications to send data immediately during the handshake itself, provided the client possesses a session ticket issued by the server during a previous QUIC connection between the two. This feature is commonly referred to as *0-RTT*.

- In TCP, network switching events (such as a mobile device moving from a mobile network to a WiFi hotspot) are handled very inefficiently: since each connection is identified by the IP address and port of each endpoint, they all become invalid and have to time out before the application can manually react and establish new connections. QUIC, on the other hand, identifies connections through IDs that uniquely identify a specific connection independently from the network path, allowing it to perform *connection migrations* between different paths.

QUIC is currently implemented in all major web browsers, including all Chromium-based browsers, Mozilla Firefox and Apple Safari, and on numerous web servers and load balancers.

Though QUIC's usage has gone hand-in-hand with HTTP/3 since its conception, it is designed as a general-purpose transport protocol and its applications extend beyond HTTP. These include secure VPN tunneling, voice over IP, media streaming and gaming.

One particularly relevant application for the purposes of this thesis is that of satellite communication:

- GEO (Geostationary Earth Orbit) satellites orbit at an altitude of 35 786 km, maintaining a fixed position relative to the Earth's surface by matching the planet's rotational period; this ensures continuous coverage and minimal link intermittency. However, due to the high altitude, GEO links are often characterized by a high RTT latency (around 600 milliseconds). TCP's three-way handshake can often represent a non-negligible performance overhead in this situation, especially for short-lived connections and when TLS is also required. QUIC's single transport/security handshake model and 0-RTT data can greatly reduce the impact of this limitation.
- LEO (Low Earth Orbit) satellites orbit at much lower altitudes (500 to 2000 km), resulting in a considerably smaller RTT latency (around 40–50 milliseconds). Their low altitude, however, forces them to orbit at much higher speeds than the Earth's rotation; as a result, LEO links are often characterized by high intermittency. The network migration feature of QUIC can partly mitigate the issue, because a recipient that is no longer accessible over a particular LEO network might become accessible over a different one.

It is worth mentioning that QUIC congestion control is designed to be independent of the main

protocol; in other words, although RFC 9002 presents a possible congestion control algorithm for QUIC based on TCP New Reno, its adoption is not mandatory, thus paving the way to a possible adoption of other solutions derived from TCP congestion control algorithms (Cubic, BBR etc.). Given its good performance on GEO links, in a companion thesis a modified version of TCP Hybla has been developed for QUIC [Cavallotti-2025]. However, it is expected that solutions based on QUIC may not be as effective as those provided by DTN in the presence of channel disruptions, as QUIC, when used end-to-end, i.e. in a TCP/IP architecture, cannot take advantage of the store-and-forward approach of the DTN architecture. That said, it is also true that QUIC could be integrated in a DTN architecture if used as a convergence layer, which is actually what was done in this thesis, as will be shown later.

1.4 Picoquic

Picoquic is a minimalist open-source implementation of the QUIC protocol licensed under the MIT license [Picoquic]. It was developed by Christian Huitema in order to be able to participate more effectively in the standardization of QUIC, and therefore it closely follows the IETF specification [Picoquic-Huitema]. It also includes several extensions to QUIC, such as multipath, i.e. the ability to utilize multiple network paths simultaneously for a single connection, and a minimal HTTP/3 implementation. It is written in C, but it also provides Rust bindings, and it uses the TLS implementation provided by a separate project called picotls, which in turn depends on OpenSSL cryptographic functions.

The C API offered by Picoquic is asynchronous: instead of providing a socket-like interface, it is based on an application-provided *stream data callback*. While TCP implementations typically buffer inbound data and let the application receive such data on-demand by means of a *read* or *receive* function, Picoquic may call the stream data callback at any time to inform about new data. In the same way, TCP implementations typically buffer outbound data and let the application write on the buffer by means of a *write* or *send* function, while Picoquic calls the stream data callback to request new data to send. This approach is advantageous for certain types of applications, especially those that send and receive unstructured streams of data (such as files), or for performance analysis; however, implementing a more socket-like (blocking) interface is convenient in other cases, such as in application protocols with struc-

tured packets, which cannot easily be split at any given offset while retaining some meaning and therefore require buffering.

Picoquic can operate in two modes:

- In foreground mode, the application directly calls the function `picoquic_packet_loop` – or a variation of it – which enters the Picoquic packet loop, effectively surrendering control of the current thread to Picoquic. This is useful:
  - for simple applications which only need to react to Picoquic events without performing any other operations in parallel;
  - for applications which manually create the threads they wish Picoquic to use.
- In background mode, the application calls the function `picoquic_start_network_thread`, which starts a separate *network thread* fully managed by Picoquic. Inside this thread, Picoquic will call `picoquic_packet_loop`.

Both in foreground and background mode, a function pointer to a *packet loop callback* must be provided by the application. This will be called when certain low-level, connection-independent events occur, such as when the Picoquic packet loop is fully started and active, or after a UDP `send`/`receive`. This callback is separate from the stream data callback described above, which is connection specific – rather than packet loop specific – and is called when higher-level events occur.

It is important to note that Picoquic is not thread-safe: Picoquic functions must not be called outside of the Picoquic-managed thread. As a result, any operation that influences the packet loop can only be executed from the packet loop callback or from the stream data callback when they are called by Picoquic. In background mode, it is possible to explicitly trigger a call of the packet loop callback from outside the thread through the function `picoquic_wake_up_network_thread`. In foreground mode, a mechanism to do so does not exist, hence the background mode is more advantageous for most applications.

Picoquic also includes a few command-line programs, such as `picoquicdemo`, a performance evaluation tool like Iperf [Iperf], albeit slightly more limited, and `picoquic_sample`, a file transfer application provided as an example to show the use of the C interface of Picoquic.

Picoquic is also capable of producing *qlog* binary files [Draft-qlog] which record the full QUIC

activity in a particular connection. These files can be analyzed and graphically rendered by external applications, such as *qvis* [qvis], which is an invaluable advantage of Picoquic for functionality and performance evaluation.

1.5 Scope of this thesis

The objective of this thesis is to develop a QUIC Convergence Layer Adapter for Unibo-BP, making use of Picoquic as the underlying QUIC implementation. This is a necessary step towards the study of QUIC suitability in DTN networks, which will start with this thesis.

QUICCL has been implemented as a part of the Unibo-BP core, like TCPCLv3, rather than as a separate module, like LTP. This allows QUICCL to make use of Unibo-BP's internal C++ IPC libraries, instead of having to use the public C API. QUICCL is designed after the Unibo-BP implementation of TCPCLv3, as their role is similar. However, as QUICCL introduces a dependency on an external project, Picoquic, the compilation of QUICCL can be disabled, if preferred by the user.

In addition to the QUICCL implementation itself, this thesis also covers the implementation of a QUICCL dissector for the Wireshark packet analyzer, which is especially useful for debugging purposes.

This thesis has been developed in parallel with two other companion theses by fellow students Valentino Cavallotti and Luca Andreetti [Cavallotti-2025; Andreetti-2025]. The three theses, all developed at the German Aerospace Center (DLR) in cooperation with the University of Bologna, compose a larger project with the ambitious goal of researching the possible usage of QUIC in satellite and interplanetary networks, either "alone", i.e. end-to-end in an IP network, or as a convergence layer in a DTN network:

- Cavallotti's thesis consists in the implementation of TCP's Hybla congestion control algorithm within Picoquic. Hybla was first designed in 2004 by Prof. Carlo Caini and Dr. Rosario Firrincieli to provide a possible solution to the RTT unfairness problem, which greatly penalizes connections with large delays, such as those via a GEO satellite [Caini-2004].
- Andreetti's thesis consists in the development of an advanced version of `picoquicdemo`,

in analyzing the performance of the other two projects and in offering support to them (all three have Picoquic in common).

The rest of this thesis is organized as follows:

- Chapter 2 describes the two currently standardized versions of the TCP Convergence Layer Adapter (3 and 4), which have been used as a base for the definition of the QUIC Convergence Layer Adapter;
- Chapter 3 describes the QUIC Convergence Layer Adapter protocol in detail and how it varies from the TCPCL;
- Chapter 4 describes the QUICCL implementation that has been developed for Unibo-BP;
- Chapter 5 describes the environments used to test the functionality of the QUICCL and the implementation of its dissector for Wireshark and briefly touches on the performance analysis results;
- Chapter 6 is devoted to conclusions.

Chapter 2 The TCP Convergence Layer Adapter

As the design of the QUIC Convergence Layer Adapter is largely inspired by that of its TCP equivalent, it is useful to examine the latter before moving on to the former in the next chapter.

The TCP Convergence Layer Adapter, usually indicated as TCPCL (by omitting the *A* for *Adapter*) allows TCP to be used as a transport protocol below the Bundle Protocol layer. As the end-to-end role of the Transport Layer is redefined by the DTN architecture, the TCP connection managed by the TCPCL links two adjacent DTN nodes (a DTN hop) instead of the endpoints as in Internet. TCP can effectively be used between DTN nodes that belong to a non-challenged subset of the interplanetary network, such as the Internet on Earth or any TCP/IP network on different planets. Note that in both cases many non-DTN IP nodes can exist between the two TCP endpoints, i.e. inside one DTN hop.

The TCPCL protocol exists in two versions:

- **TCPCL version 3**, defined in 2014 by the Internet Research Task Force [RFC7242] with BPv6 in mind; it provides basic features for transferring bundles between two nodes;
- **TCPCL version 4**, defined in 2022 by the Internet Engineering Task Force [RFC9174] with BPv7 in mind; it resolves some implementation issues present in version 3, in addition to supporting optional authentication and encryption via TLS and providing extensibility mechanisms.

Since the two versions are incompatible, and as only few Bundle Protocol implementations supported version 4 when Unibo-BP was developed in 2023, its author decided to favor compatibility by preferring version 3 and leaving the possible development of version 4 to future theses. In brief, at present the only version supported by Unibo-BP is version 3.

Both versions of the TCPCL are correctly dissected by Wireshark by the same dissector, which distinguishes between the two versions.

2.1 SDNV and CBOR

SDNV (Self-Delimiting Numeric Values), first defined in 2007 together with the Bundle Protocol in [RFC5050], and later clarified in [RFC6256], is an encoding method capable of representing arbitrary-length unsigned integers or bit strings, thus overcoming the limitations posed by traditional fixed-length fields in network protocols. Two examples where such limitations had negative consequences are the IPv4 Address Length and the TCP Advertised Receive Window. Both were worked around afterwards, but the solutions adopted (in particular NAT and IPv6 for the former) still have considerable drawbacks.

An SDNV-encoded unsigned integer consists of a series of one or more bytes, where each byte can be broken down as follows:

- The most significant bit, which is zero if this is the last byte of the sequence, or one otherwise;
- The remaining seven bits, which represent actual number data.

For example, all numbers from `0` to `127` are represented with one byte (`127` ⟶ `01111111`), as usual, while the number `128` is represented as `10000001 00000000`.

SDNV is used to represent all arbitrary-length integer numbers in version 6 of the Bundle Protocol and in most related protocols, including TCPCLv3.

Version 7 of the Bundle Protocol replaced SDNV with *CBOR* (Concise Binary Object Representation) [RFC8949], a more general-purpose format based on the JSON data model [RFC8259]; it can represent objects of any complexity, such as arrays, key-value maps or floating point numbers.

TCPCLv4 abandoned SDNV as well, but instead of adopting CBOR it simply replaced all uses of SDNV with fixed-length integers. This solution is motivated by the fact that there is no integer field in TCPCL that would realistically ever require an arbitrarily large size.

2.2 TCPCLv3

This section describes version 3 of the TCPCL [RFC7242] in more detail.

The protocol initially requires the following procedure to establish a TCPCL session:

- One node (the *TCPCL client*) establishes a TCP connection with the other (the TCPCL server);
- Each node sends a *contact header* to the other, which is used to negotiate parameters for the TCPCL session with the peer and to exchange DTN EIDs (Endpoint Identifiers), which are handed over to the respective Bundle Protocol Agents. Note that this may cause a problem when a node has two EIDs, one following the dtn scheme and one following the ipn scheme, as only one EID can be declared. Some implementations, such as DTNME, allow the user to set the scheme to be used, which is sensible but does not solve the problem of a possible mismatch between client and server, such as a server expecting a dtn EID to be advertised while the client offers its ipn EID. However, in IPN the CCSDS standard mandates the use of the sole ipn scheme, thus avoiding the problem (at least in IPN).

The contact header is composed of the following fields:

- A four-octet magic number, which is the ASCII representation of the string `dtn!`;
- A one-octet version number (set to `3` for TCPCLv3);
- Eight bits of flags, of which only the four most significant ones are used and the other four must be set to zero. The four available flags, from the least to the most significant bit, have the following meanings:
  - Request acknowledgments;
  - Request reactive fragmentation;
  - Indicate support for bundle refusal (requires the first flag to be enabled);
  - Request `LENGTH` messages;
- A two-byte keepalive interval, which proposes a number of seconds between exchanges of `KEEPALIVE` messages; this can be zero to request disabling keepalives;
- The SDNV-encoded EID length, followed by the EID of the sender of the contat header.

Once the contact headers have been exchanged, each of the two nodes independently computes the actual session parameters based on the following rules:

- Each of the first three flags is enabled if and only if it was enabled in both contact

headers;

- The `LENGTH` flag applies unilaterally for the peer(s) that enabled it (see below);
- The chosen keepalive interval is the minimum of the ones chosen by the two peers; if at least one peer requested disabling keepalives, then they are disabled.

After this, the TCPCL session is established and bundles can be transferred between the two nodes in both directions – the roles of "client" and "server" stop being meaningful at this point.

All TCPCL messages start with a one-octet message header, which is composed of:

- four bits which identify the type of message (`DATA_SEGMENT`, `ACK_SEGMENT`, `REFUSE_BUNDLE`, `KEEPALIVE`, `SHUTDOWN`, `LENGTH`);
- four bits which specify flags whose meaning depends on the message type.

In order to send a bundle, a node must send one or more `DATA_SEGMENT`s, all of which contain a header and a payload with a part of the bundle data. The header contains two flags, `start` and `end`, which are set respectively on the first and on the last segment of the bundle. Naturally, if the bundle is sent in a single segment, both flags must be set in that segment. A `DATA_SEGMENT` message contains the 8-bit header, followed by the SDNV-encoded payload length and the actual payload.

It is not possible to interleave data segments that belong to different bundles on the same TCPCL session: each bundle must wait that all segments of the previous bundle are sent before its transmission can begin. The only way to interleave bundles is to fragment them in advance at the Bundle Layer, resulting in each fragment being treated as a separate bundle. It is worth noting that multiple TCPCL sessions can be established between two nodes (each either initiated by one node or the other). Once established, all the underlying TCP connections can be used in both directions. By this approach, it is possible to simultaneously send multiple bundles between the same two nodes on parallel TCP connections. From the TCPCL perspective, however, they will belong to independent TCPCL sessions.

Before commencing a bundle transfer, and if the `LENGTH` flag was set in the contact header, the sender may send a `LENGTH` message to specify the full length of the bundle that will be sent right after, allowing the recipient to refuse it if too big or to prepare storage space. `LENGTH` messages simply contain the 8-bit header followed by the SDNV-encoded bundle length.

If negotiated through the contact headers, the receiving node sends acknowledgments to data segments (ACK_SEGMENTs), so that the sender is aware of which parts of the bundle have been received and can perform reactive fragmentation at the Bundle Layer [RFC4838] if the connection is disrupted. An ACK_SEGMENT contains the 8-bit header followed by the SDNV-encoded length up to which the bundle is being acknowledged.

If negotiated through the contact headers, the receiver can request to interrupt the transmission of a bundle at any time with a REFUSE_BUNDLE message. This is useful, for example, if it detects that it has already received the bundle, or if the bundle is too big. A REFUSE_BUNDLE message contains the usual 8-bit header followed by a 4-bit reason code. The available reason codes are:

- Unknown or unspecified;
- The receiver already has the complete bundle;
- The receiver's resources are exhausted and the sender should apply reactive bundle fragmentation before retrying;
- The receiver has encountered a problem that requires the bundle to be retransmitted in its entirety.

KEEPALIVE messages can optionally be sent at a negotiated interval in order to detect connection interruptions during idle periods. This feature is especially useful in the presence of disruption. A KEEPALIVE message is only composed of the 8-bit header alone.

Before closing its side of the connection, a node must send a SHUTDOWN message, after which it is no longer allowed to send bundles, but can still send acknowledgments and refusals. SHUTDOWN messages can optionally contain an 8-bit reason (*Idle Timeout*, *Version Mismatch* or *Busy*) and/or an SDNV-encoded reconnection delay in seconds, to indicate to the peer how long to wait before attempting another connection. The presence of each of these two optional fields can be indicated through a corresponding flag in the header. SHUTDOWN messages can also be used to refuse a contact header and close the connection during session establishment.

2.3 TCPCLv4

Version 4 of the TCPCL [RFC9174] makes several modifications to version 3 to solve some implementation issues, add features and make it more future-proof.

The session establishment procedure works as follows:

- The *active* node establishes a TCP connection with the *passive* node.
- As in version 3, the nodes exchange contact headers, but these are exclusively used to negotiate the TCPCL version and TLS support. The contact headers are in fact reduced to the 4-byte magic number (`dtn!`), an 8-bit version field (to be set to `4`) and 8 bits of flags. The only available flag is `CAN_TLS`, which each node can enable or disable to indicate whether it is capable of using TLS on top of TCP:
  - If both nodes support TLS, the TLS handshake takes place after the contact headers are exchanged; subsequently, all connection data is authenticated and encrypted. The specification also contemplates the potential existence of DTN-aware certificate authorities, which issue certificates that authenticate DTN Node Identifiers in addition to DNS hostnames and/or IP addresses. This approach will be necessary in large-scale, particularly dynamic deployments of the DTN architecture, as without Node ID authentication a rogue DTN node which can successfully authenticate its IP address could potentially impersonate another DTN node.
  - Otherwise, if both nodes deem it acceptable according to their policies, the connection continues over raw TCP.

After the procedure above, TCPCLv4 messages can be sent on the TCPCL connection. Each message begins with a one-octet message header which only encodes the message type, as opposed to TCPCLv3 which embeds the message type and the flags in the same octet. Table 2.1 contains a comparison between message types for version 4 and version 3, together with brief comments.

Table 2.1: Comparison of TCPCLv3 and TCPCLv4 message types.

V4 name	V3 name	Code	Description
`SESS_INIT`	(No equivalent)	`0x07`	Contains the session parameter inputs.
`SESS_TERM`	`SHUTDOWN`	`0x05`	Indicates a wish to terminate the session.
`XFER_SEGMENT`	`DATA_SEGMENT`	`0x01`	Transmits a segment of bundle data.
`XFER_ACK`	`ACK_SEGMENT`	`0x02`	Acknowledges the reception of a segment of bundle data.
`XFER_REFUSE`	`REFUSE_BUNDLE`	`0x03`	Indicates that the transmission of the current bundle shall be stopped.
`KEEPALIVE`	`KEEPALIVE`	`0x04`	Used to keep the TCPCL session active.
`MSG_REJECT`	(No equivalent)	`0x06`	Contains a TCPCL message rejection. Version 3 lacks this feature.
(No equivalent)	`LENGTH`	`0x06`	Contains the length (in bytes) of the next bundle. Version 4 moved this information to the Transfer Length Extension, which is included in the first `XFER_SEGMENT` of a transfer.

It can be observed that TCPCLv4 mostly kept the same message codes as version 3.

The very first message to be sent by both peers immediately after the contact headers (and optionally the TLS handshake) is `SESS_INIT`, which takes over most of the roles of the contact header in TCPCLv3. A `SESS_INIT` message contains the following fields:

- The one-octet message header;
- The keepalive interval, as a 16-bit unsigned integer;
- The segment maximum receive unit, which represents the maximum payload size of a single `XFER_SEGMENT` to be received, as a 64-bit unsigned integer;
- The transfer maximum receive unit, which represents the maximum size of a bundle (including the BP headers), as a 64-bit unsigned integer;
- The Node ID length, as a 16-bit unsigned integer, followed by the Node ID of the sender (version 3 used an SDNV for this, but 64KiB seem more than adequate even for worst case variable-length “dtn” node identifiers);

- The number of octets dedicated to session extension items, as a 32-bit unsigned integer, followed by a sequence of session extension items, i.e. optional extensions to the `SESS_INIT` message, none of which have been defined yet as of TCPCLv4; each item has the following structure:
  - 8 bits of flags; the only available flag is `CRITICAL`, which indicates whether the recipient's inability to handle this extension item should be treated as a failure and cause a termination of the TCPCL session;
  - 16 bits which represent the item type;
  - 16 bits which represent the item length, followed by the actual contents.

Note that TCPCLv4, differently from TCPCLv3, does not include flags to specify whether segment acknowledgments, reactive fragmentation and bundle refusal should be enabled, because these features are mandatory in version 4.

After the `SESS_INIT` messages are exchanged, the session parameters are negotiated similarly to TCPCLv3:

- The chosen keepalive interval is the minimum between the two;
- The segment and transfer maximum transmission unit for each peer are the respective maximum receive units of the other peer.

After this procedure, the session is established and messages can be sent both ways.

In order to send a bundle, a node must send one or more `XFER_SEGMENT` messages, which are the TCPCLv4 equivalent of TCPCLv3's `DATA_SEGMENT`s and mostly work in the same way. A `XFER_SEGMENT` contains the following fields:

- The one-octet message header;
- 8 bits of flags; the only available flags are `START` and `END`, and they have the same roles as their TCPCLv3 counterparts;
- A 64-bit unsigned integer for the transfer ID; a transfer ID identifies a bundle transfer, i.e. all the consecutive `XFER_SEGMENT`s used to transfer a specific bundle will use the same transfer ID;
- Only if this is the first `XFER_SEGMENT` of a bundle (i.e. the `START` flag is set), the number of octets dedicated to transfer extension items, followed by a sequence of transfer extension

items; these have the same function and structure as session extension items, but they apply to a specific bundle transfer.

The only available transfer extension item is the *Transfer Length Extension*; its content is a 64-bit unsigned integer containing the full length of the bundle that is being transferred. This replaces TCPCLv3's `LENGTH` messages.

- A 64-bit unsigned integer encoding the payload length, followed by the payload.

TCPCLv3's `ACK_SEGMENT`s are replaced by `XFER_ACK`s, which work in the same way. They contain the 8-bit message header, followed by 8 bits of flags (which must have the same values as those in the `XFER_SEGMENT` that is being acknowledged), the 64-bit transfer ID and the 64-bit acknowledged length.

`XFER_REFUSE` messages are the equivalent of TCPCLv3's `REFUSE_BUNDLE`; they contain the 8-bit message header, followed by an 8-bit reason code and the 64-bit transfer ID.

`KEEPALIVE` messages are the same as in TCPCLv3, and they only contain the message header.

`MSG_REJECT` messages are a new type of message that is used for troubleshooting purposes; a node sends this message as a response to an unknown or unexpected message from its peer. A `MSG_REJECT` message contains the 8-bit message header, followed by an 8-bit reason code and the 8-bit message header of the message being rejected.

Session termination is initiated by one of the two peers by sending a `SESS_TERM` message with the `REPLY` flag set to `0`, which the other peer must respond to with an identical message, but with the `REPLY` flag set to `1`. Afterwards, both endpoints may finish any in-progress bundle transfers, but must not begin or accept any new ones. A `SESS_TERM` message contains the 8-bit message header followed by 8 bits of flags (the only one of which is `REPLY`) followed by an 8-bit reason code. The ability to request a specific reconnection delay is not available in TCPCLv4; choosing a reconnection delay is instead left to implementations on the basis of the reason code.

Chapter 3 The QUIC Convergence Layer Adapter

This chapter is dedicated to a formal description of the QUIC Convergence Layer Adapter protocol. From this point onward, the protocol will be referred to as *QUICCL* (omitting the *A* for *Adapter*), except when referring to implementation details.

An effort to define a QUIC Convergence Layer Adapter has already been made in 2023 at the University of Marburg, Germany [QUICL-Marburg]. The protocol described here, however, has been designed from scratch based on TCPCLv4 and is completely independent.

3.1 Improvements to TCPCLv4

Although the protocol has been designed to be as similar as possible to TCPCL version 4, there are some important differences, listed below.

3.1.1 Removal of contact headers

The main function of the TCPCLv4 contact header is to negotiate the usage of Transport Layer Security. Once that phase is completed, the peers exchange `SESS_INIT` messages, which are used to negotiate all other session parameters. On the other hand, TLS is mandatory in QUIC, and as a matter of fact its handshake is encapsulated within the transport handshake: once the connection is established, the secure session is already active, making TLS negotiation unnecessary.

The "dtn!" magic number and the version field from the contact headers are also not needed, as QUIC makes use of TLS's Application-Layer Protocol Negotiation extension (usually shortened to ALPN) [RFC7301]. For these reasons, contact headers have been completely stripped from the QUICCL protocol, and the only message exchange necessary for session establishment is the pair of `SESS_INIT`s.

3.1.2 Bundle multiplexing

Similarly to HTTP requests, bundle transfers suffer from head-of-line blocking when serialized onto a single TCP connection: in case of losses or out-of-order TCP segments, data from a later bundle may be forced to wait for the reception of data from a previous bundle before being delivered to the TCPCL.

As there is no requirement to deliver distinct bundles in order (although it may be desirable for certain applications), QUICCL can take advantage of the multi-stream capability of QUIC mentioned in the introduction. In particular, the QUICCL sender is allowed to send different bundles in parallel on as many QUIC streams as it deems necessary, the only restriction being that segments and acknowledgments of the same bundle must all be sent on the same stream. By sending N bundles in parallel on N streams, the delivery of one bundle becomes almost independent from the problems that affect other bundles, but not completely, because QUIC congestion control covers all streams, thus a loss on one stream will reduce the aggregate transfer speed of all streams.

That said, it is expected that we can generally benefit from parallel bundle transmission. As this benefit could however be counterbalanced by the introduction of disordered bundle delivery, which, although allowed, is detrimental in some applications, parallelism should be configurable in QUICCL. The way in which this is accomplished is left to implementations.

More information on how streams are used by QUICCL is provided in 3.6.1.

3.2 QUICCL session overview

QUICCL uses the same seven message types used by TCPCLv4, namely: `SESS_INIT`, `SESS_TERM`, `XFER_SEGMENT`, `XFER_ACK`, `XFER_REFUSE`, `KEEPALIVE` and `MSG_REJECT`. However, while TCPCLv4 kept message codes identical to those in TCPCLv3, QUICCL adopts a different encoding based on the session logic.

A normal, error-free QUICCL session comprises the following phases:

- The *active node* initiates a secure QUIC connection with the *passive node*;
- Each node sends a `SESS_INIT` message on stream `0` to negotiate session parameters;

- At this point, the session is established and both nodes can send `XFER_SEGMENT`, `XFER_ACK` or `XFER_REFUSE` messages on any bidirectional QUIC stream from `1` onwards, according to the semantics described later, or `KEEPALIVE` messages on stream `0`;
- One of the two nodes sends a `SESS_TERM` message on stream `0`, to which the other responds with another `SESS_TERM` message;
- Any unfinished bundle transfers are completed;
- Both nodes close the QUIC connection.

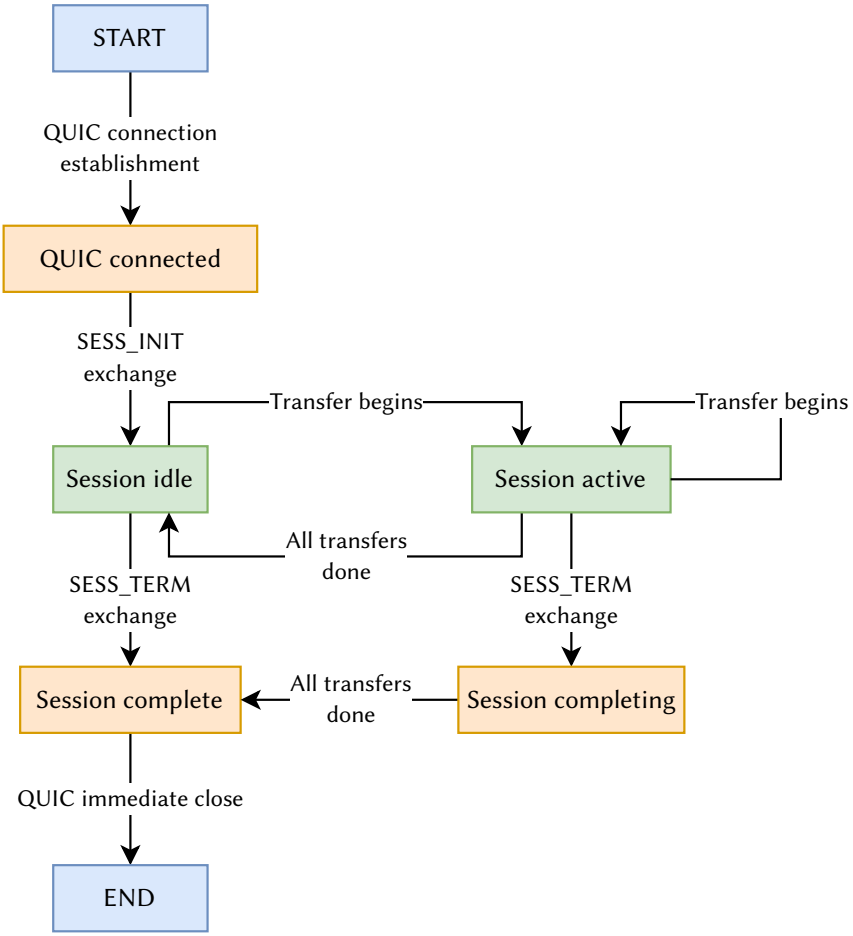

Figure 3.1: Overview of the states of a QUICCL session.

3.3 Message type header

All messages transmitted over the QUICCL session begin with a one-octet *Message Type* field. The possible values are given in Table 3.1.

Table 3.1: Available QUICCL message types.

Name	Code	Description
	`0x00`	*Reserved*
`SESS_INIT`	`0x01`	Contains the session parameter inputs from one of the entities (see 3.4)
`XFER_SEGMENT`	`0x02`	Contains a segment of bundle data (see 3.6.2)
`XFER_ACK`	`0x03`	Acknowledges a segment of bundle data (see 3.6.3)
`XFER_REFUSE`	`0x04`	Refuses a bundle transfer initiated by the peer (see 3.6.4)
`KEEPALIVE`	`0x05`	Used to keep the QUICCL session active (see 3.5.1)
`SESS_TERM`	`0x06`	Indicates a wish to terminate the session (see 3.7)
`MSG_REJECT`	`0x07`	Indicates that a message sent by the peer was not expected or understood (see 3.5.2)

The structure of each message after this field depends on the message type.

3.4 Session establishment (SESS_INIT)

As QUICCL is a connection-oriented protocol, a session must be established between the two peers before the actual exchange of bundle data. It is up to the implementation to decide when and how to trigger session establishment, however in usual cases it is expected that a DTN node which implements QUICCL should continually listen for QUICCL connections (acting as the passive node) and initiate new ones according to its configuration, as with TCPCL. A node may choose to either initiate a new session for each bundle transfer or keep sessions open for as long as possible, using them when necessary. In the former case, it is convenient to make use of QUIC's 0-RTT feature [RFC9000; RFC9308], if available.

In order to establish a QUICCL session, the active node must first establish a QUIC connection with the passive node using an appropriate QUIC implementation. We suggest using UDP port `4560` for QUICCL, but any other port can be used. The TLS Application-Layer Protocol Negotiation identifier to be used by the peers is `quicclav1`.

Once the QUIC connection is established, both peers must send a `SESS_INIT` message on

stream `0`. Since stream `0` is client-initiated according to the QUIC specification, the passive peer must receive the active peer's `SESS_INIT` before sending its own.

Each `SESS_INIT` message has the following structure, identical to its TCPCLv4 counterpart:

Name	Type
Message Type	8-bit enumeration
Keepalive Interval	16-bit unsigned integer
Segment MRU	64-bit unsigned integer
Transfer MRU	64-bit unsigned integer
Node ID Length	16-bit unsigned integer
Node ID Data	Variable-length UTF-8 string
Session Extension Items Length	32-bit unsigned integer
Session Extension Items	Octet sequence

These fields have the following meanings:

- **Message Type**: contains the value `0x01`.
- **Keepalive Interval**: contains the minimum keepalive interval, in seconds, to negotiate as the session's keepalive. It can be set to zero to indicate that no keepalive is required.
- **Segment MRU** (Maximum Receive Unit): contains the largest single-segment payload size to be received in this session.
- **Transfer MRU**: contains the largest bundle full size (i.e. bundle headers and payload) to be received in this session.
- **Node ID Length**: indicates the length, in octets, of the *Node ID Data* field.
- **Node ID Data**: contains the UTF-8 encoded DTN Node ID of the peer who sent this message. Must not be empty.
- **Session Extension Items Length**: indicates the length, in octets, of the *Session Extension Items* field.
- **Session Extension Items**: contains a sequence of extensions to this message which pertain to this QUICCL session. Each extension must follow the format defined in 3.4.1.

3.4.1 Session Extension Items

Each session extension item contained in a `SESS_INIT` message has the following structure:

Name	Type
Flags	8-bit bitmask
Type	16-bit enumeration
Length	16-bit unsigned integer
Value	Octet sequence

These fields have the following meanings:

- **Flags**: contains flags related to this Session Extension Item. The only flag defined is `CRITICAL` (`0x01`); the other fields must be set to zero by the sender and ignored by the receiver. If `CRITICAL` is set and this Session Extension Item cannot be decoded by the receiver, the QUICCL session must be terminated with a reason code of *Initialization Failure* (see 3.7). If the `CRITICAL` flag is not set and the item cannot be decoded, it must be ignored.
- **Type**: encodes the extension type. As no extensions have been defined in this version of QUICCL, there is no valid type.
- **Length**: encodes the length, in octets, of the *Value* field.
- **Value**: contains the actual extension data.

3.4.2 Session parameter negotiation

Once the `SESS_INIT` messages have been exchanged, each entity calculates internal session parameters using the following rules:

- The **Segment MTU** (Maximum Transfer Unit), which contains the largest segment payload size to be sent in this session, is set to the peer's *Segment MRU* field.
- The **Transfer MTU**, which contains the largest bundle full size to be sent in this session, is set to the peer's *Transfer MRU* field.
- The **Keepalive Interval** is set to the minimum between the *Keepalive Interval* fields in the two `SESS_INIT` messages. If at least one of them is set to zero, the keepalive functionality

is disabled for this session.

If any of these parameters turn out to be unacceptable, the entity must terminate the session with a reason code of *Initialization Failure* (see 3.7).

After the exchange of `SESS_INIT` messages, the session is fully established and other messages can be sent.

3.5 Session maintenance and status messages

3.5.1 KEEPALIVE

As mentioned in the previous section, the QUICCL protocol includes a keepalive mechanism, which an entity can use to determine whether the QUIC connection has been disrupted.

As QUIC is based on UDP, the issue of idle sessions is particularly important, because NATs and firewalls tend to drop rules for UDP ports after short intervals of inactivity, in particular much faster than for TCP, where normal rule cancellation is performed after spoofing the `FIN` exchange, while inactivity is only a backup measure. Although in certain situations QUIC connections themselves can survive NAT rebinding thanks to their reliance on connection IDs rather than on IP addresses and UDP ports, network infrastructure (e.g. load balancers) along the path may still rely on the address/port 4-tuple. [RFC9308] contains recommendations about the management of idle QUIC connections and proposes two possible solutions:

- Close a connection after an idle timeout and open it again when needed, using 0-RTT;
- Use a keepalive mechanism.

QUIC provides its own keepalive mechanism, by means of `PING` frames. Depending on the QUIC implementation, however, these may not be controllable by the application. For this reason, it was decided to keep TCPCLv4's `KEEPALIVE` messages available in QUICCL, but this feature can be safely disabled through the corresponding field in `SESS_INIT`s if the connection is kept alive through other means.

When enabled, there is no minimum value for the QUICCL keepalive interval, but it should not be too short.

KEEPALIVE messages consist of a single one-octet message header of type 0x05, with no additional data. Both sides must send a KEEPALIVE when no message has been transmitted for the negotiated keepalive interval.

KEEPALIVE messages must be sent on stream 0.

If no message of any type is received for some implementation-defined time duration, the entity must terminate the QUICCL session with a reason code of *Idle Timeout* (see 3.7). If configurable, this duration must be no less than twice the keepalive interval; if not configurable, it must be equal to twice the keepalive interval.

3.5.2 MSG_REJECT

MSG_REJECT messages must be sent by a peer when it receives a message that is either:

- *unknown*, i.e. with a message type that is deemed to be incorrect according to the implemented specification, for example due to an unhandled protocol version mismatch or an extension which adds a new message type;
- *unsupported*, i.e. with a message type that is known, but inappropriate for the negotiated session parameters, such as due to an incorrectly negotiated session extension; for example, entity A may send a session extension item without the CRITICAL flag which changes session parameters in some way; if entity B does not understand this extension, it will ignore it and send/receive invalid messages (instead, the CRITICAL flag should be set in these situations);
- *unexpected*, i.e. known and supported, but inappropriate for the current session state, for example a SESS_INIT when the session is already established, or a XFER_ACK with an unknown transfer ID.

A MSG_REJECT message must be sent on the same QUIC stream as the message it is referring to.

Each MSG_REJECT message has the following structure:

Name	Type
Message Type	8-bit enumeration
Reason Code	8-bit enumeration
Rejected Message Type	8-bit enumeration

These fields have the following meanings:

- **Message Type**: contains the value `0x07`.
- **Reason Code**: contains one of the values in Table 3.2.
- **Rejected Message Type**: contains the *Message Type* header of the message being rejected.

Table 3.2: Possible values for the *Reason code* field.

Value	Description
`0x01`	Message Type unknown
`0x02`	Message unsupported
`0x03`	Message unexpected

3.6 Bundle transfer

Bundle transfers involve the exchange of messages of three types:

- `XFER_SEGMENT`: contains a segment of bundle data;
- `XFER_ACK`: acknowledges a `XFER_SEGMENT` message;
- `XFER_REFUSE`: refuses a bundle transfer.

Each transfer may consist of one or more `XFER_SEGMENT` messages, with sizes depending on the sender's implementation and on the Segment MTU negotiated for the session.

Each bundle transfer is identified by an unsigned 64-bit *Transfer ID* field, which is contained in all `XFER_SEGMENT`, `XFER_ACK` and `XFER_REFUSE` messages pertaining to that particular transfer. Transfer IDs from each entity shall be unique within a single QUICCL session. Upon exhaustion of the 64-bit Transfer ID space, an entity must terminate the session with a reason code of *Resource Exhaustion* (see 3.7).

Transfer IDs can be freely chosen by the sender, as long as they follow the constraint specified above.

3.6.1 QUIC stream usage

A QUIC connection organizes data into streams, which are virtually not constrained in number (they are identified by a 64-bit integer) and can be either unidirectional or bidirectional. QUICCL makes use of bidirectional streams only, but in a peculiar way, which could be called logically unidirectional: data are sent only in the forward direction (i.e. from the stream initiator to the other peer), while the return direction is reserved to acknowledgments and other signaling information related to data. Incidentally, this is the same behavior as LTP sessions, thus it is not surprising at all to DTN researchers. Stream `0` is reserved to QUICCL signals concerning session initiation and management, such as `SESS_INIT`, `KEEPALIVE` and `SESS_TERM`, and cannot be used for data. Bundle transfers from the active entity to the passive entity must take place on client-initiated QUIC streams (i.e. `4`, `8`, `12` etc.); bundle transfers from the passive entity to the active entity must take place on server-initiated QUIC streams (i.e. `1`, `5`, `9`, `13` etc.). `XFER_SEGMENT` messages related to the same bundle transfer must all be sent on the same stream. `XFER_ACK` and `XFER_REFUSE` messages must be sent on the same stream used by the bundle transfer they refer to, but in the reverse direction.

A bundle transfer is considered completed by its initiator when either:

- all `XFER_SEGMENT`s have been sent and all the corresponding `XFER_ACK`s have been received, or
- a `XFER_REFUSE` has been received.

Multiple transfers can be pipelined on the same stream: it is not necessary for a bundle transfer to be fully acknowledged or refused before a new one is initiated on the same stream – it is enough that all `XFER_SEGMENT`s belonging to the previous transfer have been sent.

Entities are allowed to send a QUIC `FIN` on a stream to indicate that they will not use it to send bundles anymore.

We believe it should be left to the implementation of the sending entity to decide how many streams to use and which stream to assign each transfer to, as the receiver should be able to

de-multiplex the data in any case. The only requirement is that streams should not be wasted, to preserve resources. For example, if stream `5` exists and is currently idle, while stream `9` does not exist yet, for a new transfer the sender should prefer the reuse of stream `5` to the creation of stream `9`.

Three possible modes of operation have been considered so far in this thesis, although more could be conceived in the future. These are presented below.

- **Single stream mode**: every bundle is sent on the same stream (e.g. stream `1` for the passive entity, stream `4` for the active entity); a new bundle transfer can begin as soon as all the `XFER_SEGMENT`s for the previous transfer have been sent.

  This mode does not exploit the possibility offered by streams to send bundles in parallel, thus it suffers from the same head-of-line blocking of TCPCL; on the other hand, it accomplishes ordered delivery of bundles, which, although not compulsory, is often desired by multimedia applications.

- **N-limited multi-stream mode**: multiple streams can be used to send bundles, but their number is limited to a given integer N; every time a new bundle must be sent, one of the streams in the pool is chosen, following some policy, and the bundle is sent on it. A new bundle transfer can begin on the same stream as soon as all the `XFER_SEGMENT`s for the current transfer have been sent (different streams do not influence each other in this sense).

  This mode suffers from head-of-line blocking too, but to a lesser extent, as this phenomenon is limited to bundles sent on the same blocked stream (the others are obviously not blocked). On the other hand, it only accomplishes ordered delivery for bundles sent on the same stream. An implementation could take advantage of this by using the same stream for all bundles belonging to the same "flow" (an experimental characteristic offered by the ECOS bundle extension [Draft-ECOS]).

- **Unlimited multi-stream mode**: here the number of streams in the pool is not bounded. In contrast to the previous case, until a transfer is completed on a stream (i.e. all `XFER_ACK`s have been received or a `XFER_REFUSE` has been received), that stream is considered busy and cannot be reused for other transfers. When choosing a stream to use for a new

transfer, any non-busy one can be chosen according to some policy (e.g. the lowest-numbered one). If all existing streams are busy, a new one is created.

The goal of this last strategy is to fully prevent head-of-line blocking by sending a bundle on a stream only when the last transfer on that stream, if any, has been completed (fully acknowledged or refused by the other peer).

Like the previous one, this mode accomplishes ordered delivery only for bundles sent on the same stream. It is however not suitable for sending bundle flows, because each bundle should wait for the previous one to be fully acknowledged before being sent – a difference that can be especially relevant if the round-trip time is long.

In order to have the best of two worlds, i.e. to accomplish ordered delivery for bundle flows, while avoiding head-of-line blocking for ordinary bundles, the use of a hybrid mode may be envisaged, where N streams are reserved to the N-limited multi-stream mode and used preferentially by bundle flows, while ordinary bundles are processed in the unlimited mode.

Let us stress once again that the policies shown here are only examples and are not exhaustive: the strategy adopted by the sending entity does not matter on the receiver's side, because all a receiver has to do is to read incoming data from all streams and respond to every `XFER_SEGMENT` with a `XFER_ACK` or a `XFER_REFUSE` on the same stream.

The receiver must forward bundles to the Bundle Protocol Agent in the same order it receives them, if they belong to the same stream.

3.6.2 Data transmission (XFER_SEGMENT)

A sequence of `XFER_SEGMENT` messages is used to transfer the data of a full bundle, i.e. the primary block followed by any other blocks which form the bundle.

A single `XFER_SEGMENT` message contains a header followed by a payload, which contains a fragment of the bundle data described above.

Each `XFER_SEGMENT` message has the following structure:

Name	Type
Message Type	8-bit enumeration
Message Flags	64-bit unsigned integer
Transfer ID	64-bit unsigned integer
Transfer Extension Items Length	32-bit unsigned integer (*optional*)
Transfer Extension Items	Octet sequence (*optional*)
Data Length	64-bit unsigned integer
Data Contents	Octet sequence

These fields have the following meanings:

- **Message Type**: contains the value `0x02`.
- **Message Flags**: contains flags as laid out in table 3.3. Reserved flags must be set to zero by the sender and ignored by the receiver.
- **Transfer ID**: contains the unique identifier of this transfer.
- **Transfer Extension Items Length**: only present if the `START` flag is active, i.e. if this is the first segment for this transfer; contains the length, in octets, of the *Transfer Extension Items* field.
- **Transfer Extension Items**: only present if the `START` flag is active; contains a sequence of extensions which pertain to this bundle transfer. Each extension must follow the format defined in 3.6.2.1.
- **Data Length**: contains the length, in octets, of the *Data Contents* field of this segment (not the full bundle size).
- **Data Contents**: contains the payload of this segment.

Table 3.3: Possible `XFER_SEGMENT` flags.

Value	Name	Description
`0x01`	`END`	This is the last segment of this transfer.
`0x02`	`START`	This is the first segment of this transfer.
Others	Reserved	

3.6.2.1 Transfer Extension Items

Each Transfer Extension Item contained in a `XFER_SEGMENT` message with the flag `START` has the following structure:

Name	Type
Flags	8-bit bitmask
Type	16-bit enumeration
Length	16-bit unsigned integer
Value	Octet sequence

These fields have meanings identical to those of Session Extension Items:

- **Flags**: contains flags related to this Transfer Extension Item. The only flag defined is `CRITICAL` (`0x01`); the other fields must be set to zero by the sender and ignored by the receiver. If `CRITICAL` is set and this Transfer Extension Item cannot be decoded by the receiver, the transfer must be refused with a reason code of *Extension Failure* (see 3.6.4). If the `CRITICAL` flag is not set and the item cannot be decoded, it must be ignored.
- **Type**: encodes the extension type. The only valid type in this QUICCL version is `0x0001`, which represents the *Transfer Length Extension*, described in 3.6.2.2.
- **Length**: encodes the length, in octets, of the *Value* field.
- **Value**: contains the actual extension data.

3.6.2.2 Transfer Length Extension

The *Transfer Length Extension* is the only transfer extension defined by this QUICCL version. Its purpose is for the sender to include the full bundle size in the first segment of a transfer, so as to allow the receiver to allocate resources for the bundle that is about to be transferred, or to refuse it in case it is too large.

To include this extension, the sender must include a Transfer Extension Item with the *Type* field set to `0x0001`, the *Length* field set to `8` and the *Value* field set to an unsigned 64-bit integer containing the full bundle length.

3.6.3 Data acknowledgments (XFER_ACK)

Although the QUIC protocol provides reliable transfer of data between peers by including Transport Layer acknowledgments, it is not common for QUIC implementations to inform the application (which is represented by the QUICCL implementation in this case) of when the receiver has processed transmitted data; in other words, QUIC acknowledgments are internal to QUIC. Moreover, as a general rule, it is always preferable to have confirmations sent by the peer entity, instead of relying on those provided by the underlying protocol.

For this reason, the QUICCL protocol includes its own feedback system through `XFER_ACK` messages, which have two main purposes:

- to let the peer which initiated a bundle transfer know when the whole bundle has been successfully received, and thus when the associated QUIC stream becomes free and can be reused for a new transfer without incurring into head-of-line blocking;
- in case of a connection disruption, to let the Bundle Protocol Agent know which portion of the bundle has been successfully sent, so that it can apply reactive bundle fragmentation [RFC4838].

Each `XFER_ACK` segment has the following structure:

Name	Type
Message Type	8-bit enumeration
Message Flags	8-bit bitmask
Transfer ID	64-bit unsigned integer
Acknowledged Length	64-bit unsigned integer

These fields have the following meanings:

- **Message Type**: contains the value `0x03`.
- **Message Flags**: contains the exact same value as the *Message Flags* field in the `XFER_SEGMENT` that is being acknowledged.
- **Transfer ID**: contains the unique identifier for this transfer.
- **Acknowledged Length**: contains the cumulative payload length that is being acknowl-

edged for this transfer.

Each time an entity receives a `XFER_SEGMENT`, it must send a corresponding `XFER_ACK` after the segment has been fully processed.

The *Acknowledged Length* field must contain the sum of the *Data Length* fields of all `XFER_SEGMENT`s belonging to this transfer received until now, i.e. it is cumulative. For example, suppose the sending entity transmits four `XFER_SEGMENT` messages with lengths `100`, `200`, `500` and `1000`. The receiving entity shall send four `XFER_ACK` messages with lengths `100`, `300`, `800` and `1800` respectively, each after processing one of the `XFER_SEGMENT`s. Note that `XFER_SEGMENT`s from the same transfer, and therefore from the same stream, are delivered in order to QUICCL by QUIC (the same way as TCP does with TCPCL).

3.6.4 Transfer refusal (XFER_REFUSE)

QUICCL allows a peer to refuse the reception of a bundle before the bundle has been fully received. The choice of when to refuse a bundle is left to the implementation, and usually involves communication between the QUICCL Layer and the Bundle Layer. For example, a bundle may be refused when the Bundle Protocol Agent's bundle storage is temporarily constrained, or if a bundle header is considered unacceptable.

A `XFER_REFUSE` message may be sent in response to any `XFER_SEGMENT` message (even if the `XFER_SEGMENT` has not been fully received yet), as long as it has not been acknowledged. At its arrival, the sender associates it to the correct ongoing transfer via the *Transfer ID* field and informs the Bundle Protocol Agent of the refusal. If the reason code cannot be decoded, it should be treated as *Unknown*.

Each `XFER_REFUSE` message has the following structure:

Name	Type
Message Type	8-bit enumeration
Reason Code	8-bit enumeration
Transfer ID	64-bit unsigned integer

These fields have the following meanings:

- **Message Type**: contains the value `0x04`.
- **Reason Code**: contains one of the values in Table 3.4.
- **Transfer ID**: contains the unique identifier for this transfer.

Table 3.4: Possible values for the *Reason Code* field.

Value	Name	Description
`0x00`	Unknown	The reason is unknown or unspecified.
`0x01`	Completed	The receiver already has the complete bundle. The sender may consider the bundle as fully received.
`0x02`	No Resources	The receiver's resources are exhausted. The sender should apply reactive fragmentation before retrying.
`0x03`	Retransmit	The receiver has encountered a problem that requires the bundle to be retransmitted in its entirety.
`0x04`	Not Acceptable	Some issue with the bundle data or the transfer extension data was encountered. The sender should not retry the same bundle with the same extensions.
`0x05`	Extension Failure	A failure processing the Transfer Extension Items has occurred.
`0x06`	Session Terminating	The receiving entity is in the process of terminating the session. The sender may retry the same bundle in a different session.

When sending a `XFER_REFUSE` message, the receiving entity must have either acknowledged all previous segments pertaining to the transfer, or already refused the transfer.

After receiving a `XFER_REFUSE` message, the sender must stop sending `XFER_SEGMENT` messages for the same transfer (i.e. other parts of the same bundle), but partially sent messages must be completed before stopping.

On the other side, after sending a refusal the receiver should be ready to receive further `XFER_SEGMENT` messages for the same transfer, since these may cross "on the wire" with the refusal. In this case, the receiver should respond to each new segment with another `XFER_REFUSE` message.

3.7 Session termination (SESS_TERM)

Rather than directly closing the QUIC connection, a QUICCL entity should gracefully indicate its desire to do so to the peer. The purpose of this is to allow any existing transfers to be completed, while preventing new transfers.

To terminate a session, a peer must send a `SESS_TERM` message on stream `0` with the `REPLY` flag unset. Upon receiving the `SESS_TERM` message, the other peer must respond with an identical message, but with the `REPLY` flag set.

Each `SESS_TERM` message has the following structure:

Name	Type
Message Type	8-bit enumeration
Message Flags	8-bit bitmask
Reason Code	8-bit enumeration

These fields have the following meanings:

- **Message Type**: contains the value `0x06`.
- **Message Flags**: the only flag defined is `REPLY` (`0x01`), which has the meaning described above; all other fields must be set to zero by the sender and ignored by the receiver.
- **Reason Code**: contains one of the values in table 3.5.

Table 3.5: Possible values for the *Reason Code* field.

Value	Name	Description
`0x00`	Unknown	The reason is unknown or unspecified.
`0x01`	Idle Timeout	The session is being terminated due to idleness.
`0x02`	Busy	The entity is too busy to handle the current session.
`0x03`	Initialization Failure	The entity cannot interpret or negotiate a `SESS_INIT` option.
`0x04`	Resource Exhaustion	The entity has run into some resource limit and cannot continue the session.

A SESS_TERM message can be sent at any point in the session lifetime, but not before sending the SESS_INIT message.

Once either peer has sent a SESS_TERM message, the session is in the *Completing* state. In this state, a peer may finish any in-progress transfers, but must not initiate or accept new ones. If a transfer is received, it must be refused with a reason code of *Session Terminating*.

Once all transfers are completed, the session enters the *Complete* state. In this state, no further messages must be sent and the connection must be closed via a QUIC immediate close.

There are circumstances where an entity might wish to close the QUIC connection immediately, without waiting for transfers to complete. In this case, it must transmit a SESS_TERM message and acknowledge all XFER_SEGMENTs received until now, so as to avoid unnecessary bundle retransmissions, before closing the QUIC connection directly.

If the QUIC connection is closed uncleanly at any point during the session, any initiated transfers must be considered failed and the Bundle Protocol Agent must be notified of it. Note that [RFC9171] leaves the implementation free of reacting to this failure by resending the bundle or not. Unibo-BP, ION and other implementations do this. This, however, does not imply that the bundle is retransmitted to the same node, or by means of the same Convergence Layer, as the bundle is rerouted (e.g. by CGR/SABR) before being resent.

Chapter 4 QUIC Convergence Layer Adapter implementation

4.1 Unibo-BP's convergence layer architecture

Unibo-BP [Unibo-BP; Persampieri-2023] has been briefly presented in the Introduction. Here we will focus on the aspects of interest in building a QUIC-based Convergence Layer Adapter, i.e. QUICCL.

Unibo-BP is a highly distributed application, both multi-threaded and multi-process. The Bundle Protocol Agent is represented by a single multi-threaded process, `unibo-bp`, which communicates to other processes in the infrastructure, such as convergence layers below or applications above, via Unix sockets. Each of these processes can consist of multiple threads.

The code of Unibo-BP is organized into multiple libraries, which are accessible to all programs that form the core of Unibo-BP's architecture. One of these libraries is the CLA library, which implements a server running on the BPA and a client that represents a particular convergence layer. Each convergence layer internal to Unibo-BP (at present only TCPCLv3) can make direct use of the client part of the library; external convergence layers, such as Unibo-LTP, instead have to make use of Unibo-BP's C API. QUICCL is part of Unibo-BP (although not compiled by default so as not to introduce dependencies on Picoquic when not necessary), thus it can take advantage of the C++ API like TCPCLv3.

4.1.1 CLA server

The CLA server is tasked with communicating with all convergence layers and implementing the interface between the CLA and the BPA on the BPA's side. It allows the two entities to exchange inbound/outbound bundles and information on contacts, ranges and link openings/closures. The Bundle Protocol's ECOS fields [Draft-ECOS] are also passed from the BPA to the CLA, in order to allow the CLA to make delivery choices based on ECOS preferences. For example:

- the QUICCL could use bundle flow information to send bundles from the same flow on the same stream, as mentioned in 3.6.1;
- Unibo-LTP can select a session color (green or red) based on the ECOS `0x0008` bit, which requests reliability; more elaborate policies are also possible by considering other ECOS flags, bundle lifetimes and number of retransmissions, which are also passed to the CLA by Unibo-BP [Persampieri-2023].

The CLA server consists of a few C++ classes.

The `Link` class contains information about an outgoing link to a specific neighbor. As multiple links to the same neighbor could exist, each link has a unique identifier, used in all communications between the CLA client and the CLA server to refer to it.

The `Peer` class manages all links to a given neighbor. This class handles the extraction of bundles directed to the neighbor from the BP queue and directs them to one of the existing links towards the peer, assuming a contact is open. The way the link is selected when there are multiple possibilities is at present round-robin, but more adequate solutions are under study. The `Peer` class also applies flow control according to the data rate specified (in bytes per second) by the current contact to the neighbor.

The `CLA` class handles communication to a specific CLA client; each instance of this class is associated with an identifier that corresponds to the CLA client being handled.

4.1.2 CLA client

The Bundle Protocol's CLA library provides a `CLAOverIPC` class which abstracts the IPC mechanisms. In particular, it listens for IPC messages sent by the server (e.g. "send a bundle" or "initiate a new connection") and forwards them to the convergence layer by calling methods of the `BPToCLAController` interface, which is implemented and possibly extended by the CLA client according to its needs. Two types of requests are contemplated:

- `passthrough` requests: configuration inputs issued by the `unibo-bp-admin` command, e.g. to create an induct (i.e. start listening for incoming connections) or an outduct (i.e. initiate a new outgoing connection);
- `send_pdu`: a request by the BP to send a bundle.

4.2 QUICCL architecture

The QUICCL has been implemented after the existing TCPCL implementation. It consists of the following components, all included in Unibo-BP's CLA module:

- `quiccla_manager`: a singleton that acts as the coordinator between the Bundle Protocol Agent and the QUICCL sessions;
- `BPToQUICCLAController`: implements the `BPToCLAController` interface already mentioned, forwarding BP requests to the `quiccla_manager`;
- `quiccla_command`: contains utility methods to encode and decode IPC configuration messages sent by `unibo-bp-admin`;
- `quiccla_opportunistic_neighbor`: listens for incoming QUICCL sessions and initiates them; note that a QUICCL session initiated by a peer is bidirectional, thus it can be used "opportunistically" also in the reverse direction;
- `quiccla_planned_neighbor`: initiates new outgoing QUICCL sessions;
- `quicclav1_session`: handles all communication that pertains to a specific session, after it has been initiated via either `quiccla_opportunistic_neighbor` or `quiccla_planned_neighbor`;
- `quicclav1_util`: contains utility methods to encode and decode QUICCL messages for communications with the peer.

In addition to the QUICCL implementation itself, a Picoquic wrapper has been added to the `io` module to abstract the interface between the QUICCL and Picoquic. This wrapper contains three classes:

- `PicoQUICContext`: manages the lifetime of a Picoquic network thread and makes it possible to easily execute code within it, which is necessary to avoid data races when calling Picoquic-related functions;
- `PicoQUICSocket`: acts as an abstraction which makes Picoquic's asynchronous interface accessible via a `send`/`recv` interface similar to traditional Unix sockets;
- `PicoQUICServerSocket`: listens for incoming QUIC connections and implements an `accept` function which blocks until a connection is initiated, at which point it spawns a new `PicoQUICSocket` and continues listening for more connections.

4.3 Synchronous C++ adapter for Picoquic

Before discussing the implementation of the QUICCL protocol per se, it is appropriate to describe in detail the wrapper that was implemented for it to interact with Picoquic.

4.3.1 Picoquic's application interface

As already mentioned in the Introduction, Picoquic's interface with the application (QUICCL in this case) is asynchronous. Picoquic manages its own thread, which it has full control of, and uses it to run its packet loop, which repeatedly sends and receives QUIC packets. When the application starts the packet loop, it has to provide a pointer to a packet loop callback function, which is later called by Picoquic at each new event related to the packet loop. These are low-level events, independent of connections and mainly related to the management of the packet loop or to specific Picoquic configurations.

The packet loop callback also returns a value which Picoquic reacts to; the value `0` indicates that no error has occurred and Picoquic can proceed normally. Another possible value is `PICOQUIC_NO_ERROR_TERMINATE_PACKET_LOOP`, which requests the termination of the packet loop.

The possible data callback events are summarized in the following table.

Name	Description
`ready`	The packet loop is fully started and ready to receive packets.
`after_receive`	The packet loop has just received one or more UDP datagrams.
`after_send`	The packet loop has just sent one or more UDP datagrams.
`port_update`	The packet loop has opened the UDP socket and is informing the application of the local port being used.
`time_check`	Upon request, it is raised at every iteration of the packet loop; it provides information about the current time and the time elapsed since the previous iteration.

`system_call_duration`	Upon request, it is raised at every iteration of the packet loop; it provides the duration of the `select` system call (or another operating system's equivalent), used by Picoquic to wait for events.
`wake_up`	`picoquic_wake_up_network_thread` has been called.
`alt_port`	In case extra sockets were required to test multi-path/migration, it provides the port of the second (i.e. alternative) socket that was created.

In addition to the packet loop callback, the application can (but does not have to) provide the pointer to a connection-specific stream data callback, which Picoquic will call when events related to that connection occur. A default stream data callback can be set when initializing a Picoquic handle; later, it can be set to a different function for a particular connection by calling `picoquic_set_callback`.

The following table summarizes some of the events that trigger a stream data callback call; those not relevant to this thesis have been omitted for brevity.

Name	**Description**
`stream_data`	Data has been received on a stream.
`stream_fin`	A `FIN` has been received on a stream, optionally with data.
`stream_reset`	A `RESET_STREAM` frame has been received for a stream, which indicates an abrupt termination of the sending side of a stream.
`stop_sending`	A `STOP_SENDING` frame has been received for a stream, which indicates that any data sent on that stream will be discarded and no more data should be sent on it.
`stateless_reset`	Indicates that the connection has been terminated through a stateless reset.
`close`	Indicates that the connection has been cleanly terminated or terminated due to a QUIC error.

`application_close`	Indicates that the connection has been terminated due to an application error.
`prepare_to_send`	Represents a request by Picoquic to provide new data to send on a stream, after the application indicated its desire to do so by calling `picoquic_mark_active_stream`.
`almost_ready`	Indicates that the connection is not fully established, but data can already be sent.
`ready`	Indicates that the connection is fully established (i.e. data can be sent and received and connection migration is possible).

4.3.2 PicoQUICContext

`PicoQUICContext` is a C++ class that manages the lifetime of a Picoquic network thread. The following listing shows its declaration.

```
extern "C" int picoquic_context_packet_loop_callback(
        picoquic_quic_t *,
        picoquic_packet_loop_cb_enum cb_mode,
        void *_callback_ctx,
        void *callback_arg
);
class PicoQUICContext {
    public:
        PicoQUICContext(
                picoquic_quic_t *quic,
                std::unique_ptr<picoquic_packet_loop_param_t> packet_loop_param,
                int &ret
        );
        ~PicoQUICContext();

        void run_in_network_thread(std::function<int()> func);
    private:
        std::mutex mutex;
        picoquic_quic_t *quic{};
        picoquic_network_thread_ctx_t *thread_ctx{};
        std::unique_ptr<picoquic_packet_loop_param_t> packet_loop_param;
        std::queue<std::function<int()>> thread_queue{};
        bool thread_woken_up = false;

        void run_in_network_thread_unsafe(std::function<int()> func);
        int packet_loop_callback(
```

```
                picoquic_packet_loop_cb_enum cb_mode,
                void *callback_arg
        );
        friend int picoquic_context_packet_loop_callback(...);
};
```

When an instance of `PicoQUICContext` is constructed, it spawns the network thread by calling `picoquic_start_network_thread`, providing the Picoquic handle and the packet loop parameters chosen by the caller and setting the packet loop callback to `picoquic_context_packet_loop_callback`. `this` is passed as the callback context, to make sure that the callback has access to the `PicoQUICContext` itself.

```
PicoQUICContext::PicoQUICContext(
        picoquic_quic_t *quic,
        std::unique_ptr<picoquic_packet_loop_param_t> packet_loop_param,
        int &ret
) {
    this->quic = quic;
    this->packet_loop_param = std::move(packet_loop_param);
    this->thread_ctx = picoquic_start_network_thread(
            quic,
            this->packet_loop_param.get(),
            picoquic_context_packet_loop_callback,
            this,
            &ret
    );
    if (!this->thread_ctx) {
        THROW_UNIBO_BP_GENERIC("Failed to create picoquic network thread");
    }
}
```

After the context has been constructed, any thread can call `run_in_network_thread` with a function pointer; this method wakes up the Picoquic network thread, which executes the pointed function once it is awoken. To do this, the function pointer is added to a queue (`thread_queue`) and `picoquic_wake_up_network_thread` is called if the thread has not been woken up already.

```
void PicoQUICContext::run_in_network_thread_unsafe(std::function<int()> func) {
    thread_queue.push(func);
    if (!thread_woken_up) {
        picoquic_wake_up_network_thread(thread_ctx);
        thread_woken_up = true;
    }
```

```
}
void PicoQUICContext::run_in_network_thread(std::function<int()> func) {
    std::unique_lock lock(mutex);
    run_in_network_thread_unsafe(std::move(func));
}
```

The only role of the packet loop callback is to run any function pointers in the queue when it is woken up. If one of these functions returns an error (i.e. a non-zero value), that value is returned so that Picoquic may react accordingly.

```
extern "C" int picoquic_context_packet_loop_callback(
        picoquic_quic_t *, picoquic_packet_loop_cb_enum cb_mode,
        void *_callback_ctx, [[maybe_unused]] void *callback_arg
) {
    auto ctx = static_cast<PicoQUICContext *>(_callback_ctx);
    if (cb_mode == picoquic_packet_loop_wake_up) {
        // Execute functions requested via run_in_network_thread()
        std::unique_lock lock(ctx->mutex);
        ctx->thread_woken_up = false;
        while (!ctx->thread_queue.empty()) {
            auto func = ctx->thread_queue.front();
            ctx->thread_queue.pop();
            lock.unlock();
            if (int ret; (ret = func()) != 0)
            return ret;
            lock.lock();
        }
    }
    return 0;
}
```

When it is destructed, the `PicoQUICContext` terminates the network thread and deallocates the Picoquic handle.

```
PicoQUICContext::~PicoQUICContext() {
    std::unique_lock lock(mutex);

    /* picoquic_delete_network_thread internally terminates the thread
       before deleting the context */
    picoquic_delete_network_thread(thread_ctx);
    picoquic_free(quic);
}
```

The classes `PicoQUICSocket` and `PicoQUICServerSocket`, described below, wrap the `PicoQUIC-Context` in an `std::shared_ptr`. In this way, the network thread is automatically terminated

once all references to the context are released, i.e. once it is not being used for any active sessions (`PicoQUICSocket`) or to listen for incoming ones (`PicoQUICServerSocket`).

4.3.3 PicoQUICServerSocket

`PicoQUICServerSocket` listens on a specified UDP port for new QUIC connections. To accomplish this, it creates a `PicoQUICContext` (this starts a network thread), setting the default stream data callback for new connections to a function it manages, `picoquic_server_socket_stream_data_callback`. This function reacts to each new connection by creating a `PicoQUICSocket` for it, bound to the same network thread.

The stream data callback and the `accept` method are given below. The callback detects a new connection when it is called with the event `picoquic_callback_almost_ready`; when this occurs, it creates a new `PicoQUICSocket` and adds it to a queue of inbound connections. The accept method blocks until the queue is non-empty, then it extracts the first socket and returns it.

```
extern "C" int picoquic_server_socket_stream_data_callback(
        picoquic_cnx_t *cnx, uint64_t stream_id, uint8_t *bytes,
        size_t length, picoquic_call_back_event_t event,
        void *callback_ctx, void *stream_ctx
) {
    const auto server_socket = static_cast<PicoQUICServerSocket *>(callback_ctx);
    if (event == picoquic_callback_almost_ready) {
        // Got a new connection: create I/O socket and add it to accept queue
        std::unique_lock lock(server_socket->mutex);
        auto socket = std::make_unique<PicoQUICSocket>(
                server_socket->quic_ctx, cnx, server_socket->stoken
        );
        server_socket->inbound_cnx.push(std::move(socket));
        server_socket->inbound_cnx_cond.notify_one();
    }
    return 0;
}

std::unique_ptr<PicoQUICSocket> PicoQUICServerSocket::accept() {
    std::unique_lock lock(mutex);
    inbound_cnx_cond.wait(
            lock,
            [this] { return !inbound_cnx.empty() || _is_shutdown; }
    );
    if (_is_shutdown)
        THROW_UNIBO_BP_GENERIC("shutdown has been called on this server socket");
```

```
    auto socket = std::move(inbound_cnx.front());
    inbound_cnx.pop();
    return socket;
}
```

4.3.4 PicoQUICSocket

The `PicoQUICSocket` class is tasked with managing a specific QUIC connection. Its instances may be created in two different ways:

- *Passively*, i.e. by a `PicoQUICServerSocket` when a new connection is established; in this case, the connection's stream data callback is changed from the one managed by `PicoQUICServerSocket` to a different one managed by `PicoQUICSocket`, so that every new event related to the connection is handled by the latter rather than the former;
- *Actively*, via the `connect` static method, which initiates a new connection to a specified server address/port pair; in this case, the `connect` method creates a new `PicoQUICContext` (which starts a network thread) and sets the stream data callback to the one managed by `PicoQUICSocket`.

It is important to note that, in the passive case, the same network thread is used for all `PicoQUICSockets` created by the same `PicoQUICServerSocket`. Due to how Picoquic is implemented, this is the only option – it is not possible to make Picoquic listen for connections on a listener thread and start a dedicated thread for each new connection. By contrast, in the active case a dedicated network thread is used for each socket.

Once the connection is established, the application can interact with a `PicoQUICSocket` through several methods. The most relevant ones are listed below.

- `send`: sends data on a particular stream, taking in the stream ID and the source buffer as parameters.
- `recv`: receives data from a particular stream, taking in the stream ID and the destination buffer as parameters. If no data is available on the indicated stream, it blocks until some data is received.
- `recv_any`: it works similarly to `recv`, but it receives data on any stream; its parameters are a reference the stream ID will be written on and the destination buffer. If no data is

available on any stream, it blocks until some data is received on some stream.

- `wait_inbound`: waits for data to be available on the indicated stream ID without actually reading it.
- `wait_inbound_any`: waits for data to be available on any stream without actually reading it, and informs the caller about the stream ID.
- `send_fin`: sends a `FIN` on the indicated stream ID to inform the peer that no more data will be sent on it.
- `close`: terminates the connection through a QUIC immediate close.

The socket internally maintains one outbound and one inbound buffer for each stream. These buffers are implemented as C++ `std::deques` (double-ended queues), which are dynamically sized data structures optimized for insertion and removal of data both at the beginning and at the end.

When data is sent through the send method, it is written on the outbound buffer and the stream is marked as active via `picoquic_mark_active_stream`. This causes Picoquic to call the stream data callback with a `picoquic_callback_prepare_to_send` event. The callback then extracts the data from the outbound buffer and sends it.

Similarly, when data is received, Picoquic calls the stream data callback with the event `picoquic_callback_stream_data` or `picoquic_callback_stream_fin`. The callback reacts by copying the received data into the inbound buffer for the affected stream. When the `recv` method is called and the inbound buffer contains data, that data is returned; otherwise, `recv` waits until something is received.

4.4 Command-line interface

Unibo-BP shifts the burden of command-line argument parsing to the CLI11 library, which provides a simple yet powerful environment to configure command-line interfaces [CLI11]. This library is used by all programs within Unibo-BP, from the Bundle Protocol Agent itself to its convergence layers and included applications; as a consequence, it has been used for the QUIC Convergence Layer as well.

The command-line interface of the QUICCL is very similar to that of Unibo-BP's TCPCL; more

specifically, a user of Unibo-BP interacts with QUICCL in two ways:

- First through the `unibo-bp-quiccl` executable, to start the QUICCL process;
- Once started, through the `quiccl` sub-command of the `unibo-bp-admin` command (i.e. `unibo-bp-admin quiccl`), to communicate with the running QUICCL process, for example with the goal of configuring it or stopping it.

The `unibo-bp-quiccl` command supports the following options:

- `--daemon`: runs QUICCL as a daemon rather than in the current terminal. When this option is used, the PID (process identifier) of the daemon is printed to the standard output at its start.
- `--cla-id <id>`: allows the user to choose a different identification number for the CLA. This may be useful if multiple CLAs use the same identifier by coincidence; in addition, it allows running multiple QUICCL instances on the same machine, though this latter scenario is not particularly useful, as one can start multiple QUICCL sessions on the same instance regardless.

`--cla-id` is also an option of the `quiccl` subcommand of `unibo-bp-admin`; in this case, it allows the user to select the CLA to administer.

In addition to this, `unibo-bp-admin quiccl` supports the following command tree:

- `induct`: this command is used to administer inducts. An induct (the term has been borrowed from ION's terminology) is a server listening on a UDP port for incoming connections; typically, a node does not require more than one induct, but adding multiple ones can be useful to listen on multiple ports. The only subcommand of `induct` is `add`, which is used to start an induct. Hence, the full command to be used to start an induct is `unibo-bp-admin quiccl induct add`. The `--port` option can be used to specify the UDP port to listen on.
- `outduct`: this command is used to administer outducts. An outduct (again, borrowed from ION's terminology) is a client that connects to an induct located on another DTN node, thus creating a DTN hop between them. The only subcommands of `outduct` are `add` and `remove`; `add` requires the following options to identify the peer to connect to:
  - `--hostname <ip_address>`, where `ip_address` is an IPv4/IPv6 address in the typ-

ical ASCII notation;

- `--port <port>`, where `port` is the UDP port the peer's induct is listening on (the default is `4560`);
- `--peer <node_id>`, where `node_id` is the DTN Node ID of the peer.

For example, the command to be used to start a QUICCL session with the node `ipn:6.0` located at the IP address `10.0.3.6` is:

```
unibo-bp-admin quiccl outduct add --peer ipn:6.0 --hostname 10.0.3.6
```

The `remove` command only requires the `--peer` option to identify the outduct to be removed.

- `stop`: this command stops the QUICCL process.

In addition to those listed above, the `induct` and `outduct` commands both accept the following options to tweak their behavior:

- `--single-stream`: instructs the QUICCL to send all bundles on a single QUIC stream; the default is to work in an unlimited multi-stream mode (see 3.6.1 for details);
- `--congestion-control <id>`: instructs Picoquic to use a specific congestion control algorithm among the ones it supports (e.g. `bbr`, `reno`, `cubic`, `hybla`, etc.); the default is `bbr`;
- `--qlog <path>`: instructs Picoquic to output qlog files in the specified directory, recording QUIC activity on connections used by the QUICCL;
- `--textlog <path>`: instructs Picoquic to output a text log file at the specified path, useful for debugging purposes;
- `--keylog <path>`: instructs Picoquic to output a TLS secret log at the specified path, necessary to decrypt QUIC traffic on packet analyzers like Wireshark (see 5.1.2).

When adding an outduct, these options affect the QUIC connection that will be used for that outduct; when adding an induct, they affect all QUIC connections initiated by peers towards that induct. It is important to note that the `--single-stream` flag is unidirectional, i.e. it only affects how the local node sends bundles, not the peer. To make both nodes in a link use a single stream, both the induct and the outduct that connects to it must have the `--single-stream` flag enabled.

4.5 QUICCLA manager

`quiccla_manager` is the class that manages all existing QUICCL sessions, handling requests coming from the Bundle Protocol Agent and from the sessions themselves. In particular, when the BPA makes a request via inter-process communication, it is received by the class `BPToQUICCLAController`, which in turn calls a function in the QUICCLA manager; by contrast, QUICCLA sessions call QUICCLA manager methods directly.

The commands that can be issued by the BPA are the following:

- `register_planned_neighbor`: corresponds to the addition of a new outduct and results in the creation of a `quiccla_planned_neighbor` (see 4.6);
- `deregister_planned_neighbor`: corresponds to the removal of an existing outduct;
- `spawn_inducts`: corresponds to the addition of a new induct and results in the creation of a `quic_listener_thread` (see 4.6);
- `send_pdu`: corresponds to a request to send a bundle using a specific QUICCL session;
- `stop`: corresponds to a request to stop the QUICCL.

QUICCLA sessions make use of the following methods to retrieve information or to inform the manager and/or the BPA of certain events:

- `get_administrative_endpoint`: retrieves the Node ID of this node;
- `insert_session`: informs the manager and BPA that a new session has been started – either actively through an outduct or passively through an induct – and can be used to send bundles;
- `remove_session`: informs the manager and the BPA that a previously existing session no longer exists and cannot be used to send bundles;
- `notify_transmission_success`: informs the BPA that a bundle has been sent successfully, i.e. completely acknowledged by the peer;
- `notify_transmission_failure`: informs the BPA that a bundle has not been sent successfully;
- `notify_inbound_pdu`: informs the BPA that a new bundle has been received and delivers it accordingly.

4.6 Planned and opportunistic neighbors

The terms *planned neighbor* and *opportunistic neighbor* refer to QUICCL sessions where the local node is active and passive respectively. The rationale is that, while an active node knows in advance when it is going to start a session, a passive node waits for inbound sessions with no prior knowledge of when they are going to be initiated.

At its construction, the `quiccla_planned_neighbor` class starts a new thread that actively initiates a QUICCL session (using a PicoQUICSocket) and waits until it is terminated. Afterwards, it assesses whether it is appropriate to try initiating the session again; if not, it terminates.

The `quiccla_opportunistic_neighbor` module contains the `quic_listener_thread` function, which continually listens for new connections (using a `PicoQUICServerSocket`) and passively initiates a QUICCL session whenever an inbound connection is detected.

Both the `quiccla_planned_neighbor` and the `quiccla_opportunistic_neighbor` are responsible for the entire QUICCL session establishment procedure, i.e. the initiation of a QUIC connection followed by the `SESS_INIT` message exchange and the computation of the resulting session parameters. After the session is established, in order to deal with its normal operation, they both start a new `quicclav1_session` (for QUICCL version 1) and add it to the manager. In case future QUICCL versions are designed and implemented in Unibo-BP, during session establishment the planned neighbor and the opportunistic neighbor should detect the version in order to select the appropriate implementation.

4.7 QUICCLAv1 session

The `quicclav1_session` class manages a specific QUICCL session after it has been initiated. Since QUICCL sessions are almost symmetrical after they are established (the only asymmetrical aspect being stream numbering), it is not necessary to handle active and passive connections separately.

Whenever the BPA requests to send a bundle, the manager's `send_pdu` method is called, which in turn calls `send_pdu` on the appropriate QUICCL session. This method adds the bundle to a queue of outbound PDUs for that session.

At its creation, a QUICCLA session starts one sender thread and one receiver thread. The sender thread repeatedly waits for new bundles in the outbound queue and sends them, while the receiver thread repeatedly receives messages from the peer and reacts to them accordingly.

Depending on whether the `--single-stream` command-line flag is enabled, the sender thread may adopt either the "single stream" or the "unlimited multi-stream" policy, both described in 3.6.1:

- If the single stream flag is enabled, the first stream allowed by the QUICCL protocol is always chosen, thus if the local node is the active peer it always sends on stream 4, while if it is the passive peer it always sends on stream 1.
- Otherwise, multiple streams are used: each stream has a boolean `busy` state associated to it; when sending, the lowest-numbered non-busy stream is chosen and marked as busy (if none exists, a new one is created); once the bundle is fully acknowledged or refused by the peer, the receiver thread marks the stream as non-busy, so it can be used by the sender for a new bundle.

When a bundle is sent, a new entry is added to a per-stream acknowledgment queue, which is shared between the sender and the receiver. This entry contains information about acknowledgments related to that bundle, most importantly its full length, the sent length and the acknowledged length. When the receiver thread receives an `ACK_SEGMENT`, it updates the acknowledged length, and once the bundle is fully acknowledged the entry is removed.

The receiver thread, on the other hand, always reads inbound data from all streams, as the protocol requires.

4.8 Build system

The build tool used by Unibo-BP is CMake [CMake], which facilitates cross-platform compilation. On the basis of platform-agnostic rules, declared by a project's developers in files called `CMakeLists.txt`, CMake automatically configures platform-specific build tools, which in turn build the source code. This allows a project's codebase to be built on multiple platforms without writing platform-specific rules.

For example, the default native tool used by CMake on GNU/Linux is GNU Make [Make], which builds a project's codebase according to the rules written in a `Makefile`. When CMake is invoked on a project, it reads the `CMakeLists.txt` files and generates a `Makefile` accordingly, so that the project can then be built with Make. On Windows, one can instruct CMake to generate Visual Studio project files instead, without changing the `CMakeLists.txt` files at all.

As mentioned in the introduction, QUICCL has been implemented as an optional module in Unibo-BP, because it requires Picoquic as a dependency, a useless overhead at compile time if the user is not interested in QUICCL. When the CMake variable `ENABLE_QUICCL`, which is off by default, is enabled, QUICCL source files and code blocks are added to their respective CMake targets. For example, the source files in `src/io/socket/quic` are only compiled if the flag is enabled.

In order to download and compile Picoquic from its Git repository in a way that is transparent to the user, we took advantage of the CMake module FetchContent [CMake-FetchContent]. This is particularly effortless in Picoquic's case, because it also uses CMake.

Picoquic in turn depends on the picotls library, but it is also capable of using FetchContent internally to obtain it if the CMake flag `PICOQUIC_FETCH_PTLS` is enabled.

The following listing shows the `CMakeLists.txt` file used to fetch Picoquic.

```
if (WITH_QUICCL)
    include(FetchContent)
    FetchContent_Declare(picoquic
            GIT_REPOSITORY https://github.com/MAC-Projects/picoquic.git
            GIT_TAG origin/master
    )

    # Make sure Picoquic fetches picotls:
    set(PICOQUIC_FETCH_PTLS ON CACHE BOOL "" FORCE)
    FetchContent_MakeAvailable(picoquic)

    set(pico_targets picoquic-core picoquic-log picotls-core picotls-fusion
            picotls-minicrypto picotls-openssl)

    foreach(target IN LISTS pico_targets)
        set_target_properties(${target} PROPERTIES POSITION_INDEPENDENT_CODE ON)
    endforeach()

    find_package(OpenSSL REQUIRED)
```

```
    add_library(third-party-picoquic INTERFACE)
    target_include_directories(third-party-picoquic INTERFACE
            ${picoquic_SOURCE_DIR}/picoquic ${picoquic_SOURCE_DIR}/loglib)
    target_link_libraries(third-party-picoquic INTERFACE
            ${pico_targets}
            OpenSSL::SSL OpenSSL::Crypto
    )
endif()
```

This code performs the following actions:

- It declares the Picoquic Git repository, specifying a branch. If necessary, it is also possible to fetch a specific commit – this is generally advised, because changes in the dependency would otherwise require changes in the dependent code; on the other hand, if both projects are in development and moving quickly, it is convenient to directly use a Git branch.
- It sets the flag `PICOQUIC_FETCH_PTLS`, so that Picoquic will in turn fetch the picotls library, which is its own dependency.
- It makes the Picoquic library (and, by extension, picotls) available to the project.
- It declares an array (`pico_targets`) which contains a list of the libraries that compose Picoquic and picotls and enables position independent code on all of them, which is generally necessary for libraries.
- It finds the OpenSSL package, which is required by Picoquic as well.
- It creates a new CMake library, called `third-party-picoquic`, in `INTERFACE` mode, meaning that it is not a library per se, rather it is only a container for compile-time information.
- It configures `third-party-picoquic` so that it is associated with Picoquic's include directories, then it links `third-party-picoquic` to Picoquic's and its dependencies' library files.

In this way, a CMake target can be linked to Picoquic and made to use its include directories with a single CMake command. The only Unibo-BP target where this has been necessary is the `io` library, which contains the classes `PicoQUICSocket`, `PicoQUICServerSocket` and `PicoQUICContext`, the only ones using Picoquic directly:

```
if(WITH_QUICCL)
```

```
    target_link_libraries(io PUBLIC third-party-picoquic)
endif()
```

Chapter 5 Wireshark dissector and functional evaluation

5.1 Wireshark dissector

In order to make it easy to debug and analyze packets containing QUICCL traffic, a Wireshark dissector for the QUIC Convergence Layer protocol has been developed [QUICCL-Wireshark]. This section is dedicated to a description of Wireshark, its plugin system and how the QUICCL dissector was implemented.

5.1.1 Wireshark

Wireshark is a cross-platform, free and open-source graphical packet analyzer written in C++, licensed under the GNU GPLv2 [Wireshark]. A packet analyzer is a tool that can read packet capture files, that is, files that record network frames that traverse a particular interface over a defined period. These files can be generated by Wireshark itself (in this case the tool can also analyze them in real time) or by external packet capturers or analyzers like `tcpdump` [tcpdump].

More specifically, Wireshark reads the full content of each frame and shows it in a human-readable form, separating headers that belong to different encapsulated protocols and describing the fields they contain. The packets are shown in a list, and clicking on one displays its details on the bottom-left of the window.

For example, imagine a packet containing a non-encrypted HTTP response has been captured. When it is selected, Wireshark displays a series of expandable menus, each representing a different network layer. Expanding these menus reveals detailed information extracted from the corresponding headers. In this case, the resulting sequence of menus might look like the following:

- Raw frame contents, with general information about the frame that is not held inside it, such as its arrival timestamp;
- Ethernet header (e.g. source and destination MAC addresses);

- IPv4 header (e.g. source and destination IP addresses);
- TCP header (e.g. source and destination port, sequence number);
- HTTP header (e.g. status code, content type);
- HTTP payload.

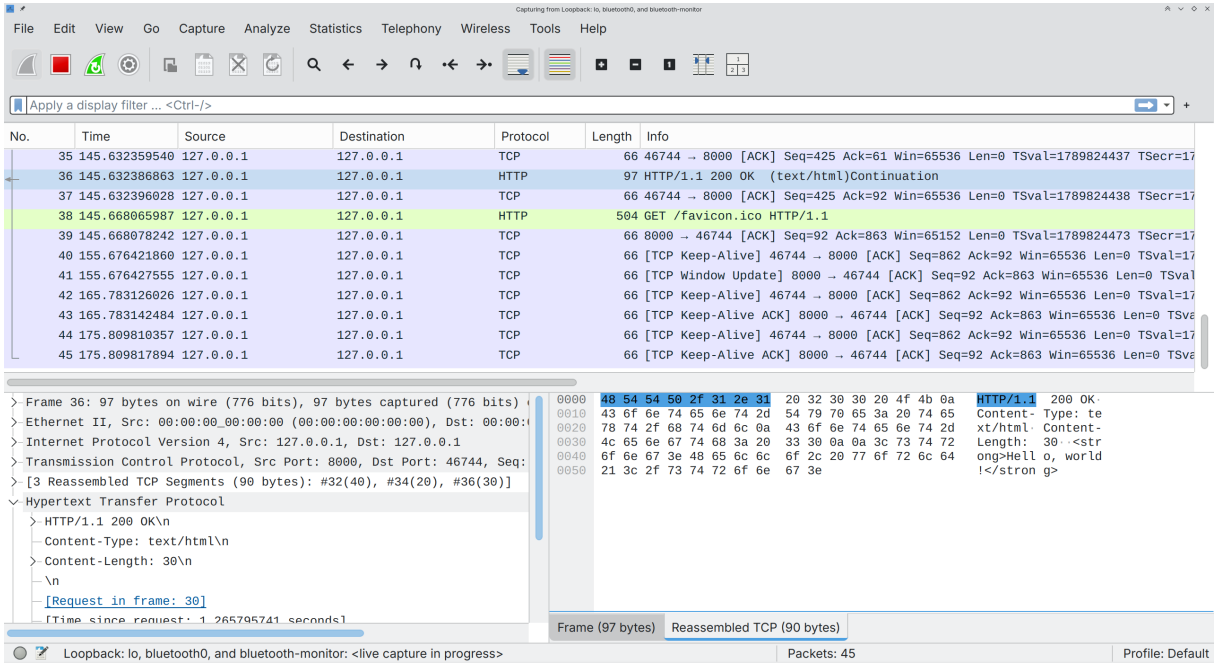

Figure 5.1: Wireshark dissection of an HTTP response: notice the protocol tree on the bottom left.

The components of Wireshark responsible for reading header contents and generating this human-readable information are called *dissectors*. There is a Wireshark dissector for nearly every standardized network protocol, including QUIC and the Bundle Protocol.

Each dissector is responsible for decoding the header of a particular protocol, after which it may call another dissector to decode the encapsulated protocol, or offer a mechanism for dissectors to register based on matching rules (for example, dissectors can indicate to the TCP dissector that they should be used for connections directed to a specific port). This creates a call stack, where each dissector outputs the information it retrieves, optionally calls another dissector and then returns to the caller. For each network frame, the first dissector that is called is always the *frame dissector*, which displays general information about the frame and calls an appropriate Data-Link dissector (e.g. Ethernet) on the basis of the frame contents.

A very large number of dissectors is included by default in Wireshark, under the path `epan/`

`dissectors` of the source tree; usually, a dissector is a simple C source file which includes the following functions, where `protoname` stands for an identifier of the protocol being dissected:

- `dissect_protoname(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree, void *data)`: dissects the frame portion which contains data of the protocol at hand; the parameters have the following meaning:
  - `tvb`: a *testy, virtual buffer* containing the protocol data (*testy* means that it checks for out-of-bounds errors; *virtual* means that its data is not necessarily physically contiguous in memory);
  - `pinfo`: general information about the packet, shared between all dissectors in the stack;
  - `tree`: a handle to the GUI tree where information about the packet should be added;
  - `data`: optional data received from the dissector of the protocol which encapsulates this protocol;
- `proto_register_protoname(void)`: is responsible for registering the protocol and the dissecting function in Wireshark's internal data structures;
- `proto_reg_handoff_protoname(void)`: defines under which circumstances this dissector should be called; for example, an HTTP dissector should be called when a TCP connection on port `80` is captured.

To add a new dissector to the Wireshark code, one should call its implementation file `packet-protoname.c`, place it into the Wireshark source tree under `epan/dissectors`, and add it to `epan/dissectors/CMakeLists.txt`. Then one should recompile Wireshark, resulting in a separate Wireshark build with the new dissector.

Unless the goal is for the dissector to be included in the official Wireshark distribution, one might prefer to write a dissector in the form of a Wireshark plugin, to avoid the recompilation of Wireshark (it takes several minutes on a fast, modern PC) and to be able to distribute the client standalone. In addition to the functions listed above, a plugin must contain the following ones:

- `uint32_t plugin_describe(void)`: returns an enumeration value describing the type of plugin; in the case of a dissector, it simply has to return `WS_PLUGIN_DESC_DISSECTOR`;
- `void plugin_register(void)`: calls `proto_register_plugin` passing in a `proto_plugin`

structure, which contains function pointers to the `proto_register_protoname` and `proto_reg_handoff_protoname` functions.

When building a plugin, the result is a shared object file (`.so` on Linux, `.dll` on Windows, `.dylib` on macOS), which can be distributed to users, who must then include it in their global or user-specific Wireshark plugin directory. The default global location is `/usr/lib/wireshark/plugins/<wireshark_version>`, while the default local one is `/.local/lib/wireshark/plugins/<wireshark_version>`.

There are two ways to make a Wireshark plugin:

- The more traditional way is to insert the plugin source file directly in the Wireshark tree, under `plugins/epan/<plugin_name>`. Although the built plugin will be a shared object file separate from the main Wireshark build, this solution still requires rebuilding Wireshark to build the plugin.
- An alternative approach, which has become available more recently, is to create the plugin out-of-tree as a separate project, so that it can be compiled by itself. This requires the Wireshark development files to be installed on the system, either through a Linux distribution package manager or by installing them from the Wireshark source code via CMake.

The QUICCL dissector, described in the following subsection, has been implemented in this second way.

5.1.2 The QUICCL dissector

For the QUICCL protocol to be recognized correctly by Wireshark, it has to be properly registered in the function `proto_register_quiccl`:

```
void proto_register_quiccl(void) {
    proto_quiccl = proto_register_protocol(
            "DTN QUIC Convergence Layer Protocol",
            "QUICCL",
            "quiccl"
    );
    proto_register_field_array(proto_quiccl, hf_quiccl, array_length(hf_quiccl));
    proto_register_subtree_array(ett, array_length(ett));

    [...]
```

```
    quiccl_handle = register_dissector("quiccl", dissect_quiccl, proto_quiccl);

    [...]
}
```

Similarly to how dissectors above TCP can register based on a port, dissectors above QUIC can register based on a TLS Application-Layer Protocol Negotiation Identifier. The following code, placed in `proto_reg_handoff_quiccl`, binds the QUICCL dissector to connections with ALPN IDs equal to `quicclav1`:

```
dissector_add_string("quic.proto", "quicclav1", quiccl_handle);
```

This, together with the previous listing, ensures that, whenever a QUIC frame containing QUICCL data is found, the QUICCL portion of it is passed to the function `dissect_quiccl`. Starting from this foundation, two factors need to be taken into consideration:

- Depending on the policy adopted by the QUIC implementation when sending packets, a single QUIC frame may contain one or more QUICCL messages, or, conversely, a QUICCL message may be split into multiple QUIC frames;
- Bundles are usually spread into multiple QUICCL messages, unless they are very small.

This means a reassembly mechanism has to be implemented in multiple layers: on the lower layer, to reassemble QUICCL messages; on the upper layer, to reassemble bundles which will later be handed over to the Bundle Protocol dissector. Both problems can be solved by using features offered directly by Wireshark.

In order to reassemble QUICCL messages, the QUICCL dissector can communicate with the QUIC dissector by setting the `desegment_offset` and `desegment_len` fields of the `pinfo` (packet info) structure. When `dissect_quiccl` is called for the first packet in a connection, it tries to detect the length of the first QUICCL message:

- If the length cannot be determined because the segment is too short, the QUICCL dissector sets `pinfo->desegment_offset` to the offset of the beginning of the message in the frame and `pinfo->desegment_len` to `DESEGMENT_ONE_MORE_SEGMENT`. By doing this, the QUIC dissector will call the QUICCL dissector again at the next QUIC frame of the

same stream, providing a buffer that contains the previous frame's data followed by the new frame's data, in hope that the message length can now be determined; if not, the QUICCL dissector keeps responding with `DESEGMENT_ONE_MORE_SEGMENT` until the length can be determined.

- If the length can be determined, but the current frame does not contain the full QUICCL message (i.e. the message length is larger than the frame length), the dissector sets `pinfo->desegment_offset` to the beginning of the message and `pinfo->desegment_len` to the message length that was determined. By doing this, the QUIC dissector will call the QUICCL dissector again when the full message has been received.
- If the length can be determined and the current frame contains the full QUICCL message, the dissector dissects the message. If a new QUICCL message is contained after the dissected one in the same frame, the length check is performed again for that message; otherwise, the dissector will be called again with the next frame and the length check will be performed then.

This safely ensures that the QUICCL dissector reassembles full QUICCL messages and can dissect them one by one. Once a message has been reassembled, the function `dissect_quiccl_message` is called, which takes care of the dissection itself.

Dissecting data in Wireshark is essentially a repetition of the following steps:

- Read a field from the testy virtual buffer at a certain offset;
- Create an item for that field in the protocol tree and/or modify other items;
- Potentially add information to the Info column in the packet list, or so-called *expert info*, which is highlighted in the tree, to signal exceptions or other inferred information.

Below an example of reading the Transfer ID (a 64-bit unsigned integer) from a QUICCL `XFER_SEGMENT` and adding it to the protocol tree:

```
uint64_t transfer_id = tvb_get_uint64(tvb, offset, ENC_BIG_ENDIAN);
proto_tree_add_uint64(message_tree, hf_transfer_id, tvb, offset, 8, transfer_id);
offset += 8;
```

In order to reassemble a bundle from multiple `XFER_SEGMENT`s, every time the QUICCL finishes dissecting the header of a `XFER_SEGMENT`, it uses the Wireshark functions `fragment_add_seq_`

`next` and `process_reassembled_data`, providing a unique identifier to make sure that segment payloads belonging to the same bundle are associated to each other:

```
uint32_t reassembly_transfer_id = (uint32_t)transfer_id * 2 +
                                  ((ctx->tx_peer == ctx->convo->active) ? 1 : 0);

fragment_head *frag_msg = fragment_add_seq_next(
        &transfer_reassembly_table, tvb, data_offset, pinfo,
        reassembly_transfer_id, &transfer_id, data_length_clamp,
        !(flags & QUICCL_TRANSFER_END_FLAG)
);
ctx->transfer_payload = process_reassembled_data(
        tvb, data_offset, pinfo, "Reassembled QUICCL transfer", frag_msg,
        &transfer_fragment_items, NULL, message_tree
);
```

The last parameter of `fragment_add_seq_next` informs Wireshark about whether this is the segment of this bundle. Once the full bundle is received, `ctx->transfer_payload` will contain a testy virtual buffer with the full bundle to be passed to the Bundle Protocol dissector, which can then add its own dissection results to the protocol tree:

```
call_dissector(bundle_handle, ctx->transfer_payload, pinfo, proto_tree);
```

As QUIC integrates TLS security, for the QUICCL dissector to function properly the QUIC dissector has to be able to decrypt traffic. To configure this, the user should navigate to `Edit -> Preferences -> Protocols -> TLS` and choose an appropriate method to provide Wireshark with the session keys. The QUICCL implementation in Unibo-BP can be configured to output a Master-Secret log file by using the `--keylog` parameter when creating an induct or an outduct (see 4.4).

An example of the QUICCL dissector in use is shown in Figure 5.2.

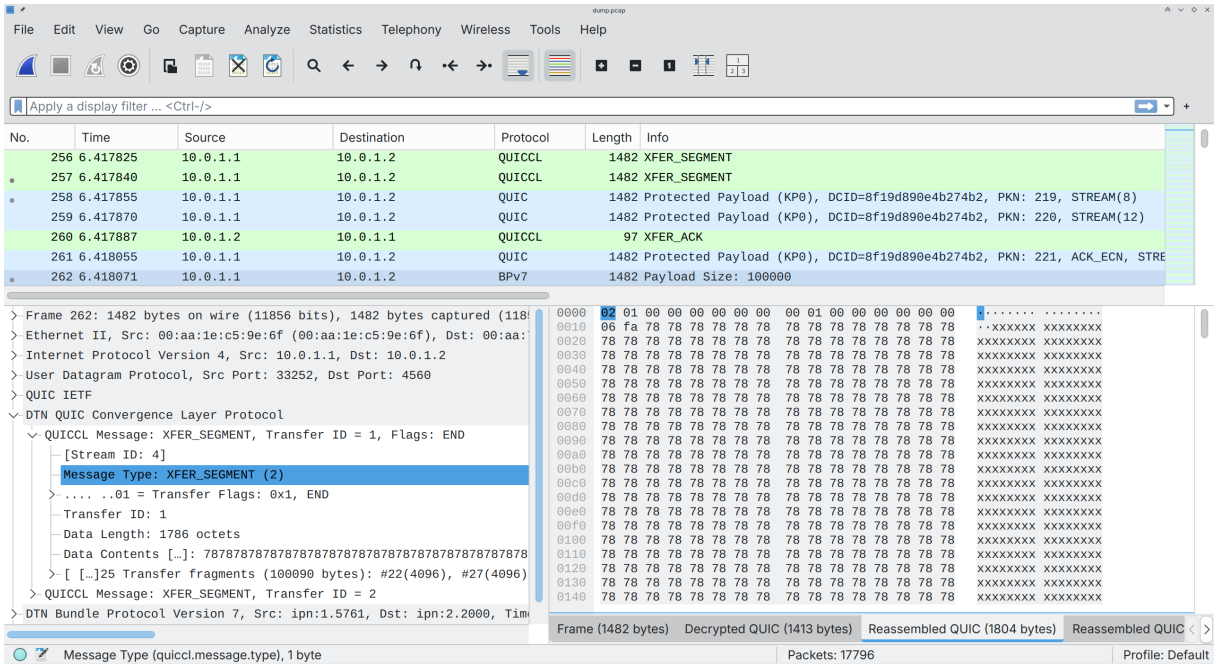

Figure 5.2: The QUICCL dissector in use. The selected frame contains the end of a bundle transfer, thus the Bundle Protocol dissector was called.

5.2 Functional evaluation

In order to test the functionality of the QUICCL implementation included in Unibo-BP, two environments have been used:

- A single-machine environment, taking advantage of Unibo-BP's feature that allows multiple DTN nodes to be hosted on the same node;
- A virtual testbed created by using Virtualbricks, a virtualization tool specialized in the arrangement of virtual networks.

This section is devoted to a description of these environments.

5.2.1 Single-machine environment

The Unibo-BP daemon relies on a few Unix sockets written under the `.unibo-bp` subdirectory of the directory from which it is started. DTN applications that want to connect to this daemon must either be started in this directory or configured to use a different directory by means of the `UNIBO_BP` environment variable. The default mechanism makes it very easy to create multiple nodes on the same machine; firstly it is necessary to create a subdirectory for each

node; then a daemon must be started in each directory; lastly, applications started in one of these directories automatically connect to the daemon started in the same directory.

In Unibo-BP's Git repository [Unibo-BP], the `config_scripts/n_nodes` directory contains some configuration files to start local testing environments that use an arbitrary number of nodes. These environments are controlled by the use of four scripts with self-explanatory names: `start-scenario`, `stop-scenario`, `force-stop-scenario` and `cleanup-scenario`. Each of these scripts takes one argument: the path to the scenario's configuration directory. This directory is structured as follows:

- It contains a subdirectory for each node, each with the following scripts, which are run when starting the scenario:
  - `start-deamon.sh`: starts the Unibo-BP daemon;
  - `induct.sh`: configures CLA inducts;
  - `outduct.sh`: configures CLA outducts;
  - `service.sh`: starts necessary application-layer programs;
- It contains a `nodes.txt` file with a list of the node directories, one per line;
- It contains a `common` directory with the following scripts, which are run in each node when starting the scenario:
  - `cla.sh`: starts the necessary CLAs;
  - `region.sh`: configures regions;
  - `extension.sh`: if needed, configures Unibo-BP extensions such as RGR and CGRR;
  - `routing.sh`: if needed, configures routing parameters;
  - `contact-plan.sh`: configures contacts.

At present, the only preconfigured environment, `sc-2-hops`, uses three nodes connected to each other in a row, forming two DTN hops. The convergence layer used in this environment is TCPCL. In order to perform simple QUICCL tests, an environment named `sc-2-hops-quic` was created.

This scenario is identical to the previous one, the sole difference being the convergence layer. As a consequence, the only configuration worth showing is that related to convergence layers.

The `cla.sh` script simply starts QUICCL and adds its PID to the `pids` file, used by other scripts

to stop the scenario afterwards:

```
#!/bin/bash
unibo-bp-quiccl --daemon | awk '{print $2}' >> pids
```

The `induct.sh` script is identical in each node (except for the UDP port), and simply starts an induct:

```
#!/bin/bash
unibo-bp-admin quiccl induct add --port 4561
```

The `outduct.sh` script for the first node starts an outduct towards the second node, logging TLS keys for potential Wireshark dissection:

```
#!/bin/bash
unibo-bp-admin quiccl outduct add \
        --peer ipn:2.0 --hostname localhost --port 4562 --keylog keys.log
```

The `outduct.sh` script for the second node starts an outduct towards the third node:

```
#!/bin/bash
unibo-bp-admin quiccl outduct add \
        --peer ipn:3.0 --hostname localhost --port 4563 --keylog keys.log
```

Since QUICCL sessions are bidirectional, with this configuration it is possible to send bundles between any pair of these nodes in either direction, using the second node as a DTN router for communications between the first and the third.

This testing environment has certain benefits and drawbacks:

- On the one hand, it is easy to deploy and to configure, making it an ideal option to verify proper operation of QUICCL during development;
- On the other hand, it is quite limited: all traffic is effectively looped back on the network layer, making it almost impossible to emulate real-world channels (e.g. channels with a low bandwidth or high delay).

5.2.2 Virtual testbed

Virtualbricks [Virtualbricks] is a free software tool for Linux machines that makes it possible to configure virtual testbeds composed by an arbitrary number of virtual machines connected

in a network configuration. Virtualbricks mainly uses two technologies under the hood:

- **QEMU**: a very well-known virtualization ecosystem, which also supports hardware virtualization, making it capable of reaching near-native speeds;
- **VDE**: a set of programs, developed by Prof. Renzo Davoli from the University of Bologna, which provide virtual Ethernet controllers; these are used by Virtualbricks to connect the virtual machines to each other by setting up switches and channel emulators.

Depending on the configuration, Virtualbricks machines can be accessed by SSH, but the usual graphical output is also supported.

Virtualbricks testbeds can be saved as "projects", which can be copied and deployed on a different machine in a very easy way. For these tests, a testbed named DTN3hops was specifically created; it can be freely downloaded from [CNRL].

The layout of this testbed, shown in Figure 5.3, contains four machines in line (VM1, VM2, VM4, VM6), forming three hops, and a fifth machine (VM3) in the same network as VM1 and VM2. VM5 is omitted because older versions of the testbed had four machines with two hops and the fifth machine was represented by the host.

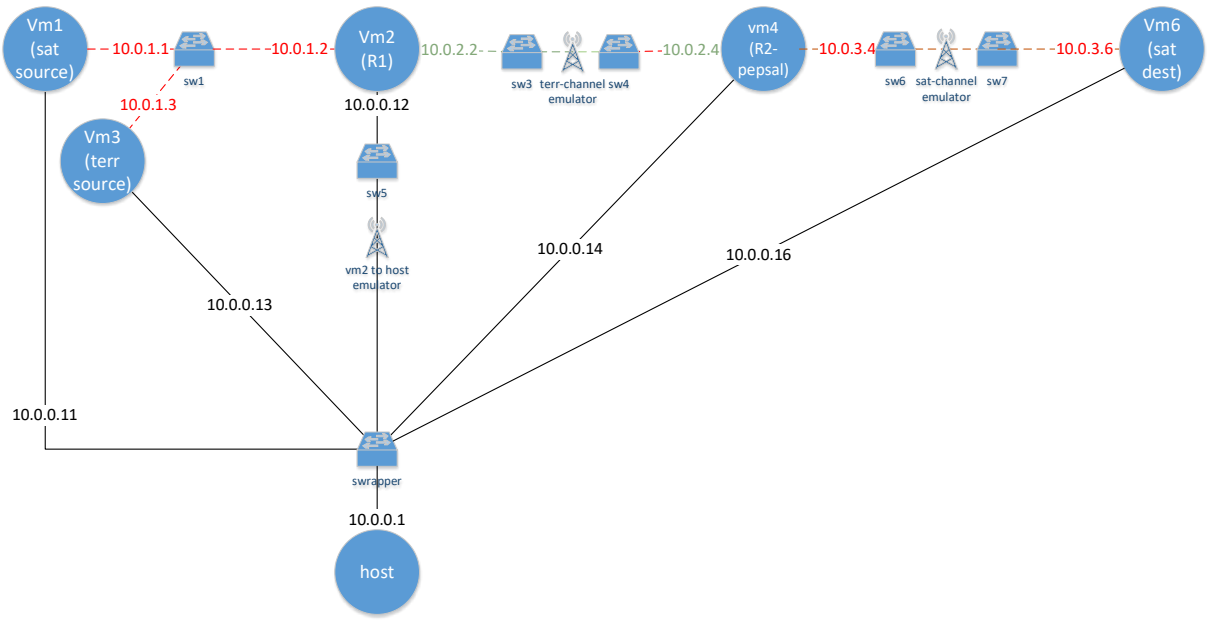

Figure 5.3: Layout of the DTN3hops testbed.

Unibo-BP with QUICCL was installed and configured on all of these machines, using a number

of DTN configurations over time, including but not limited to the following:

- The only two DTN nodes are VM1 and VM6, while VM2 and VM4 act as simple IP nodes, resulting in a single DTN hop;
- VM1, VM2, VM4 and VM6 are all DTN nodes, resulting in three DTN hops;
- VM1, VM3 and VM6 are DTN nodes and the DTN hops are formed by the pairs VM1–VM3 and VM3–VM6.

Unibo-BP is configured on each of these machines by means of a single file named `start.sh`. A possible configuration for VM2 is the following (configurations irrelevant to QUICCL have been omitted):

```
#!/bin/bash

unibo-bp start --set-storage-size 2000000000 \
        --dtn-admin dtn://vm2.dtn --ipn-admin ipn:2.0 \
        --daemon

[...]

unibo-bp-tcpcl --daemon
unibo-bp-quiccl --daemon

[...]

unibo-bp-admin tcpcl induct add
unibo-bp-admin quiccl induct add

unibo-bp-admin quiccl outduct add --peer ipn:1.0 --hostname 10.0.1.1
unibo-bp-admin quiccl outduct add --peer ipn:3.0 --hostname 10.0.1.3
unibo-bp-admin quiccl outduct add --peer ipn:4.0 --hostname 10.0.2.4
```

This file makes VM2 open three outducts towards VM1, VM3 and VM4, resulting in three bidirectional QUICCL sessions (those machines can send bundles to VM2 as well). An alternative configuration could involve outducts from VM1, VM3 and VM4 to VM2 instead, with similar results.

After a problem with the bandwidth limitation features of the channel emulator of Virtualbricks (`vde-netemu`) was discovered (it introduced disorder on frames), we decided to use the network emulator included in Linux, `tc netem` [tc-netem], to emulate real-world properties. The `netem` emulator allows the user to apply a series of challenges to outbound packets on a

certain interface, such as limited rate, delay, loss, duplication and corruption. For example, the following command sets a rate of 10Mb/s, a delay of 500ms and a 1% packet loss on the interface `eth0`, replacing any existing `netem` rules:

```
tc qdisc replace dev eth0 root netem rate 10Mbit delay 500ms loss 1% limit 10000
```

The `limit` parameter specifies how many packets `netem` should be able to hold in its queue, dropping incoming ones when the queue is full (*drop tail policy*).

A virtualized testing environment like this complements the testbed based on a single machine used during QUICCL development, as it is able to emulate a real-world scenario much better, but at the price of a higher complexity.

5.3 Performance analysis

The Virtualbricks testbed just described was used to carry out a thorough performance analysis of QUICCL in conjunction with Luca Andreetti. This section contains a brief introduction to the scenarios and some results for demonstration purposes only; for a more exhaustive treatment, the reader is referred to Luca Andreetti's thesis [Andreetti-2025].

Following the suggestion of our co-supervisor, we considered scenarios where a spacecraft sends data to a ground station, with the following characteristics:

- Data rate: 100Mb/s;
- Round-trip time: 10ms, 125ms, 2s;
- Packet error rate: 10%, 1%, 0.1%, 0.01%;
- Contact duration: persistent and with interruptions of duration 10ms, 50ms, 100ms, 1s, 10s, 100s, 200s; the first four are considered unplanned interruptions (i.e. the link is down, but the contact is up), the last three are planned (i.e. both the link and the contact are down).
- Number of streams: single stream, unlimited streams;
- Configurations:
  - For all scenarios: no data can be sent to the ground station during the interruption;
  - In addition to the configuration above, only for scenarios with planned interrup-

tions: during the interruption, the ground station is reachable by routing the bundles through another spacecraft.

The performance tests themselves were conducted using the tool DTNperf [DTNperf], an Iperf-like performance evaluation tool for the Bundle Protocol developed at the University of Bologna. DTNperf also supports bundle status reports, which have been used to log information about each test.

The graphs in Figures 5.4 and 5.5 show comparisons of single-stream and multi-stream QUICCL when sending a burst of 10 bundles in two different environments: in both cases the channel has a bandwidth of 100Mb/s, but while in the first case the transmission rate is artificially limited to 10Mb/s, in the second case it is constrained by competing UDP traffic at 90Mb/s.

In both environments, the difference between single-stream and multi-stream QUICCL can be clearly observed: when a single stream is used, the bandwidth is fully dedicated to one bundle at a time, allowing the bundles to reach the destination gradually and in order. In contrast, with multiple streams, the entire burst is delivered nearly simultaneously, leading to a longer average delivery time and potential disorder (the latter did occur in these experiments, though not visually obvious). Note that in the second graph the time axis is in a different scale, highlighting the impact of losses introduced by competing traffic, which cause severe latency in bundle delivery.

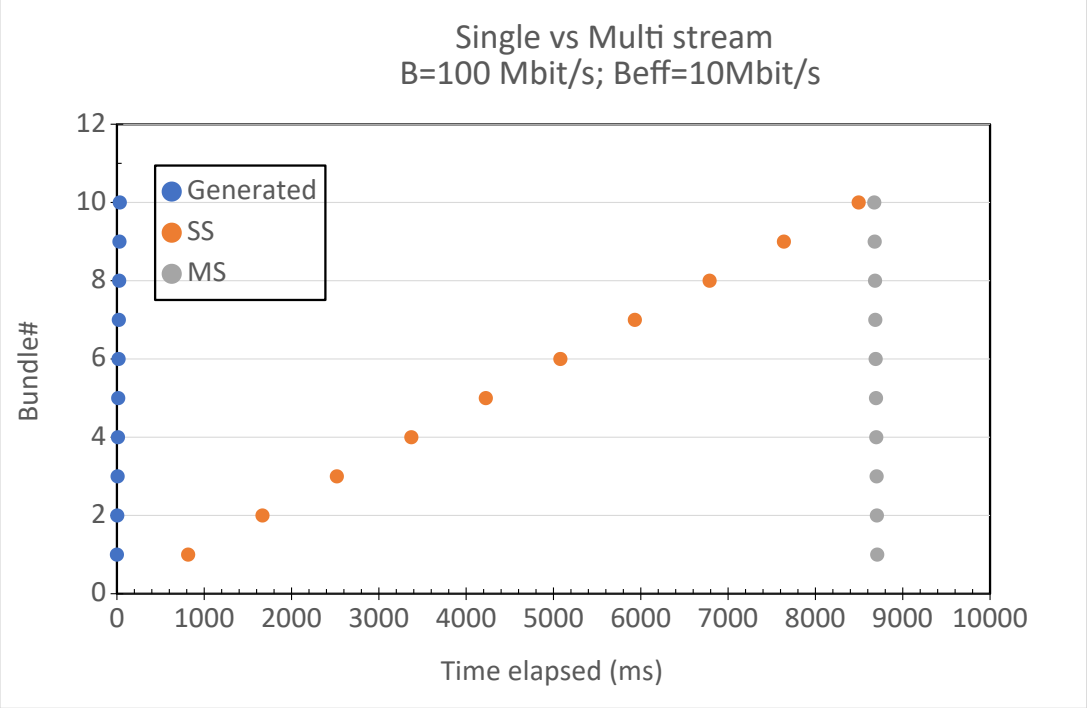

Figure 5.4: Bundle transmission and reception – comparison of single-stream and multi-stream QUICCL on a channel with 100Mb/s bandwidth and 25ms RTT, sending at 10Mb/s.

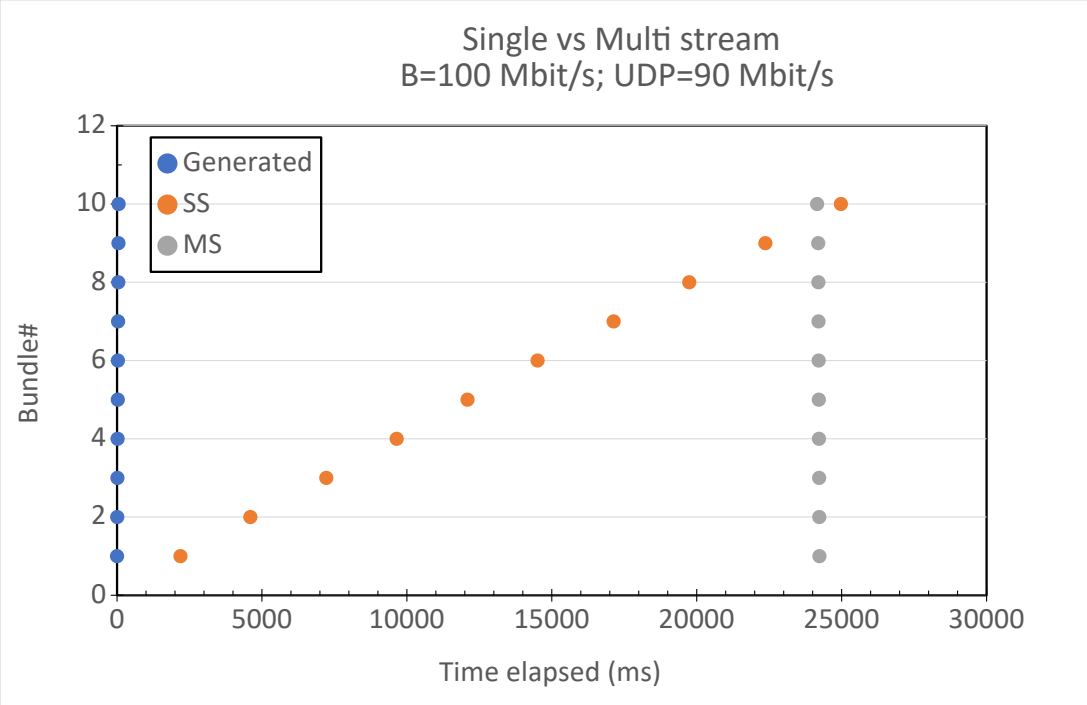

Figure 5.5: Bundle transmission and reception – comparison of single-stream and multi-stream QUICCL on a channel with 100Mb/s bandwidth and 25ms RTT, competing with UDP traffic at 90Mb/s.

Chapter 6 Conclusions

The objective of this thesis was to design and implement a new protocol, the *QUIC Convergence Layer Adapter*, to enable communication between two Delay-Tolerant Network (DTN) nodes over a QUIC connection. This document presented the complete process of research, design and implementation of the protocol. The project was carried out at the German Aerospace Center (DLR) in cooperation with the University of Bologna, addressing topics of interest to both institutions.

The work began with an in-depth study of the existing TCP Convergence Layer Adapter, which solves a similar problem, analyzing both of its versions. The focus then shifted to the QUIC protocol, with particular attention to its distinguishing features compared to TCP. Based on this research, the QUICCL protocol was designed as an adaptation of TCPCL that leverages the full capabilities of QUIC.

Following the design phase, the internal mechanisms of the Unibo-BP and Picoquic libraries were examined in detail to make the most of the implementation environment. QUICCL was developed on the basis of the existing TCPCL implementation, following common software engineering principles.

To facilitate analysis of the developed product, a dedicated Wireshark dissector was developed. This required preliminary research into existing dissectors followed by an implementation phase.

Finally, the product was evaluated in terms of functionality and performance, utilizing both built-in features of Unibo-BP and external tools. The results confirmed that the implementation meets its initial objectives: QUICCL operates correctly and its performance is in line with expectations.

The program has been released as part of Unibo-BP under the GNU GPLv3 license, in hope that it will be useful for future research.

Bibliography

[Andreetti-2025] L. Andreetti, "QUIC performance evaluation in satellite networks: development of tools and result analysis", Master's thesis, University of Bologna, Mar. 2025.

[Caini-2004] C. Caini and R. Firrincieli, "TCP Hybla: a TCP enhancement for heterogeneous networks", in: *International Journal of Satellite Communications and Networking* 22.5 (2004), pp. 547–566, DOI: https://doi.org/10.1002/sat.799, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sat.799.

[Cavallotti-2025] V. Cavallotti, "Design and implementation of a QUIC congestion control designed after TCP Hybla", Master's thesis, University of Bologna, Mar. 2025.

[CCSDS-734.2-B-1] *CCSDS Bundle Protocol Specification*, Issue 1, CCSDS 734.2-B-1, Blue Book, Sept. 2015, URL: https://public.ccsds.org/Pubs/734x2b1.pdf.

[CLI11] *Source code of CLI11*, URL: https://github.com/CLIUtils/CLI11.

[CMake] *Official website of CMake*, URL: https://cmake.org/.

[CMake-FetchContent] *Official documentation of CMake FetchContent*, URL: https://cmake.org/cmake/help/latest/module/FetchContent.html.

[CNRL] *CNRL*, University of Bologna, URL: http://cnrl.deis.unibo.it/.

[Draft-ECOS] S. Burleigh and F. Templin, *Bundle Protocol Extended Class of Service (ECOS)*, Internet-Draft draft-burleigh-dtn-ecos-00, Work in Progress, Internet Engineering Task Force, May 2021, 9 pp., URL: https://datatracker.ietf.org/doc/draft-burleigh-dtn-ecos/00/.

[Draft-IPN-update] R. Taylor and E. J. Birrane, *Update to the ipn URI scheme*, Internet-Draft draft-ietf-dtn-ipn-update-14, Work in Progress, Internet Engineering Task Force, Sept. 2024, 31 pp., URL: https://datatracker.ietf.org/doc/draft-ietf-dtn-ipn-update/14/.

[Draft-qlog] R. Marx, L. Niccolini, M. Seemann, and L. Pardue, *qlog: Structured Logging for Network Protocols*, Internet-Draft draft-ietf-quic-qlog-main-schema-10, Work in Progress, Internet Engineering Task Force, Oct. 2024, 57 pp., URL: https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-main-schema/10/.

[DTNperf] C. Caini, A. d'Amico, and M. Rodolfi, "DTNperf_3: A further enhanced tool for Delay-/Disruption- Tolerant Networking Performance evaluation", in: *2013 IEEE Global Communications Conference (GLOBECOM)*, 2013, pp. 3009–3015, DOI: 10.1109/GLOCOM.2013.6831533.

[Iperf] *Official website of Iperf*, URL: https://iperf.fr/.

[Make] *Official website of GNU Make*, URL: https://www.gnu.org/software/make/manual/make.html.

[Persampieri-2023] L. Persampieri, "Unibo-BP: an innovative free software implementation of Bundle Protocol Version 7 (RFC 9171)", Master's thesis, University of Bologna, Feb. 2023, URL: https://amslaurea.unibo.it/id/eprint/27740/.

[Picoquic] *Source code of Picoquic*, URL: https://github.com/private-octopus/picoquic.

[Picoquic-Huitema] C. Huitema, *Picoquic documentation*, URL: https://www.privateoctopus.com/picoquic.html.

[QUICCL-Wireshark] *Source code of the QUICCL dissector plugin for Wireshark*, URL: https://gitlab.com/mattiamoffa/quiccl-wireshark.

[QUICL-Marburg] M. Sommer, A. Sterz, M. Vogelbacher, H. Bellafkir, and B. Freisleben, "QUICL: A QUIC Convergence Layer for Disruption-tolerant Networks", in: *Proceedings of the Int'l ACM Conference on Modeling Analysis and Simulation of Wireless and Mobile Systems*, MSWiM '23, Montreal, Quebec, Canada: Association for Computing Machinery, 2023, pp. 37–46, ISBN: 9798400703669, DOI: 10.1145/3616388.3617525, URL: https://doi.org/10.1145/3616388.3617525.

[qvis] *Official website of qvis*, URL: https://qvis.quictools.info/.

[RFC3986] T. Berners-Lee, R. T. Fielding, and L. M. Masinter, *Uniform Resource Identifier (URI): Generic Syntax*, RFC 3986, Jan. 2005, DOI: 10.17487/RFC3986, URL: https://www.rfc-editor.org/info/rfc3986.

[RFC4838] L. Torgerson, S. C. Burleigh, H. Weiss, A. J. Hooke, K. Fall, D. V. G. Cerf, K. Scott, and R. C. Durst, *Delay-Tolerant Networking Architecture*, RFC 4838, Apr. 2007, DOI: 10.17487/RFC4838, URL: https://www.rfc-editor.org/info/rfc4838.

[RFC5050] K. Scott and S. C. Burleigh, *Bundle Protocol Specification*, RFC 5050, Nov. 2007, DOI: 10.17487/RFC5050, URL: https://www.rfc-editor.org/info/rfc5050.

[RFC6256] W. Eddy and E. B. Davies, *Using Self-Delimiting Numeric Values in Protocols*, RFC 6256, May 2011, DOI: 10.17487/RFC6256, URL: https://www.rfc-editor.org/info/rfc6256.

[RFC7242] M. Demmer, J. Ott, and S. Perreault, *Delay-Tolerant Networking TCP Convergence-Layer Protocol*, RFC 7242, June 2014, DOI: 10.17487/RFC7242, URL: https://www.rfc-editor.org/info/rfc7242.

[RFC7301] S. Friedl, A. Popov, A. Langley, and E. Stephan, *Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension*, RFC 7301, July 2014, DOI: 10.17487/RFC7301, URL: https://www.rfc-editor.org/info/rfc7301.

[RFC8259] T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC 8259, Dec. 2017, DOI: 10.17487/RFC8259, URL: https://www.rfc-editor.org/info/rfc8259.

[RFC8949] C. Bormann and P. E. Hoffman, *Concise Binary Object Representation (CBOR)*, RFC 8949, Dec. 2020, DOI: 10.17487/RFC8949, URL: https://www.rfc-editor.org/info/rfc8949.

[RFC9000] J. Iyengar and M. Thomson, *QUIC: A UDP-Based Multiplexed and Secure Transport*, RFC 9000, May 2021, DOI: 10.17487/RFC9000, URL: https://www.rfc-editor.org/info/rfc9000.

[RFC9001] M. Thomson and S. Turner, *Using TLS to Secure QUIC*, RFC 9001, May 2021, DOI: 10.17487/RFC9001, URL: https://www.rfc-editor.org/info/rfc9001.

[RFC9002] J. Iyengar and I. Swett, *QUIC Loss Detection and Congestion Control*, RFC 9002, May 2021, DOI: 10.17487/RFC9002, URL: https://www.rfc-editor.org/info/rfc9002.

[RFC9171] S. Burleigh, K. Fall, and E. J. Birrane, *Bundle Protocol Version 7*, RFC 9171, Jan. 2022, DOI: 10.17487/RFC9171, URL: https://www.rfc-editor.org/info/rfc9171.

[RFC9174] B. Sipos, M. Demmer, J. Ott, and S. Perreault, *Delay-Tolerant Networking TCP Convergence-Layer Protocol Version 4*, RFC 9174, Jan. 2022, DOI: 10.17487/RFC9174, URL: https://www.rfc-editor.org/info/rfc9174.

[RFC9308] M. Kühlewind and B. Trammell, *Applicability of the QUIC Transport Protocol*, RFC 9308, Sept. 2022, DOI: 10.17487/RFC9308, URL: https://www.rfc-editor.org/info/rfc9308.

[tc-netem] *netem – Network Emulator*, Unix manual page, URL: https://man7.org/linux/man-pages/man8/tc-netem.8.html.

[tcpdump] *Official website of tcpdump*, URL: https://www.tcpdump.org/.

[Unibo-BP] *Source code of Unibo-BP*, URL: https://gitlab.com/unibo-dtn/unibo-bp.

[Unibo-DTN] *Project directory of Unibo-DTN*, URL: https://gitlab.com/unibo-dtn.

[UnifiedAPI] A. Bisacchi, C. Caini, and S. Lanzoni, "Design and Implementation of a Bundle Protocol Unified API", in: *2022 11th Advanced Satellite Multimedia Systems Conference and the 17th Signal Processing for Space Communications Workshop (ASMS/SPSC)*, 2022, pp. 1–6, DOI: 10.1109/ASMS/SPSC55670.2022.9914734.

[Virtualbricks] P. Apollonio, C. Caini, M. Giusti, and D. Lacamera, "Virtualbricks for DTN Satellite Communications Research and Education", in: *Personal Satellite Services. Next-Generation Satellite Networking and Com-*

munication Systems, ed. by I. Bisio, Cham: Springer International Publishing, 2016, pp. 76–88, ISBN: 978-3-319-47081-8.

[Wireshark] *Official website of Wireshark*, URL: https://wireshark.org/.

Acknowledgments

First of all, I would like to express my deepest gratitude to my parents for always supporting me in my choices, both economically and emotionally, without whom I wouldn't have been able to undertake this course of study.

I must also extend my appreciation to my friends Luca and Valentino, both for making the academic workload feel lighter during these five years at the University of Bologna and for sharing these six months in Munich.

Finally, I can't but be thankful to all the nice people I met in Germany, in particular the students and employees at DLR for their support and friendliness during this experience. Special thanks go to my co-supervisor Tomaso de Cola for his assistance in the development of this project.

Mattia Moffa