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

QUIC performance evaluation in satellite networks: development of tools and result analysis

Candidate

Luca Andreetti

Supervisor

Prof. **Carlo Caini**

Co-supervisor

Dott. Ing. **Tomaso de Cola**

Academic Year 2023/2024

Abstract

The current state of global connectivity is a product of many years of research and innovations, reaching a point where the majority of the planet is able to communicate very easily in a worldwide web. Protocols such as TCP or IP are what allowed this incredible spread. There are some assumptions that need to be true for this suite of protocols to work efficiently, for example: an end-to-end path must exist for the whole duration of the communication, the round trip time must be short, and the loss rate of packets should be relatively low. As soon as at least one of these requirements is missing, the network is referred to as *challenged*, as the TCP/IP suite becomes inadequate.

The first case in which this limitation was encountered is in the filed of space communications. In this environment, there is a really high propagation delay due to the distance between nodes and the inevitable intermission of the links between them, caused by the movement of satellites and the rotation of the planets themselves. In order to take care of these restrictions, a group of researchers started working, at the beginning of the years 2000s, on the creation of a new network architecture, called *Inter-Planetary Network* (IPN). Soon after, the researchers realized that the space was not the only field which has hostile conditions to the utilization of TCP/IP, and, because of this, the challenged networks were expanded to include a broader number of environments that have strong conditions of isolation. The name was changed to *Delay-/Disruption-Tolerant Networking* (DTN) and it is largely explained in RFC 4838. This document explains how a new layer, called *Bundle Layer*, can be added between the transport and application layers. Inside of this newly introduced level, the *Bundle Protocol*, described in RFC 5050 (version 6) and RFC 9171 (version 7), is the agent that manages the sending and receiving of messages, called *bundles*. Another interesting component of the DTN architecture is what is called a *convergence layer*. This element has the important job of allowing the Bundle

Protocol to send bundles on a vast variety of networks, without having to interact directly with the underlying transport protocols. It does so by using an interface called *Convergence Layer Adapter* (CLA), and there is a specific version of it for each protocol that is intended to be used underneath the Bundle Protocol. Obviously someone has to develop it first.

Another possibility when dealing with high propagation delays is to modify the standard behavior of TCP itself. This is exactly what was done by the University of Bologna, whose researchers developed a new version of TCP that aimed at boosting the performance, and at preventing the phenomenon known as "bandwidth starvation". The name of this implementation is Hybla.

It is in this context that the project of this thesis was brought to life. This document is part of a broader research project that includes two companion theses, also developed at the German Aerospace Center (DLR), by fellow students Mattia Moffa and Valentino Cavallotti. The main objective was to investigate the possibility of using a fairly recent protocol called QUIC in satellite and interplanetary networks. In particular, the colleagues had the important job of developing a convergence layer adapter for QUIC and to adjust the implementation of TCP Hybla to the specifications of this new protocol. This thesis is focused on the description of the QUIC protocol and its implementations, the enhancement of the tools provided for testing, and the to analyze the results obtained. In particular, this document contains the description of the development of a time oriented mode and a logging system for the program used for testing QUIC as a peer-to-peer tool. The tests have been performed using a testbed of virtual machines accessible via ssh, and through a testing script that iterates over the different configurations of interest. Finally, the results have been injected using a Python script inside .ods LibreOffice files that automatically draw the graphs from the data.

Contents

Chapter 1

Introduction

Today's Internet connectivity has reached immense success, allowing the exchange and communication of data around the world. A great part of Internet success is due to the clever design of the TCP/IP protocols. In Internet's architecture, IP packets travel independently from each other from source to destination, passing through intermediate nodes called *routers*. The IP protocol is fairly simple: it is not reliable (since it does not confirm the reception of data, nor does it recover possible losses), it does not guarantee ordered delivery, and it does not implement either congestion or flow control. Sometimes this is exactly what is needed by applications, which in this case use UDP as a transport protocol on top of IP, only adding port multiplexing and an optional integrity check of segments. However. there are scenarios in which it is necessary to have a reliable service at the transport layer, with congestion and flow control: in these cases, the TCP protocol is the best choice. There are some essential conditions for this protocol to work correctly [Farrell_2011]:

- the end-to-end path must exist for the whole duration of the communication session;
- the Round Trip Time is short;
- packet loss between nodes is relatively low.

The Delay-/Disruption-Tolerant Networking architecture is conceived to relax most of these assumptions, providing a valid alternative when operating in networks that do not achieve adequate functioning of TCP and, more generally, of the TCP/IP suite.

These networks are called *challenged networks* and posses at least one of the following characteristics (as shown in Figure 1.1) [Warthman_2015]:

- the absence of an end-to-end path between source and destination (known as *network partitioning*) creates intermitted connections that render TCP/IP ineffective;
- long propagation and variable queuing delays contribute to end-to-end path intermissions that can defeat protocols and applications that rely on quick return of acknowledgements;
- high bit error rates, due to interferences or other problems on the channel.

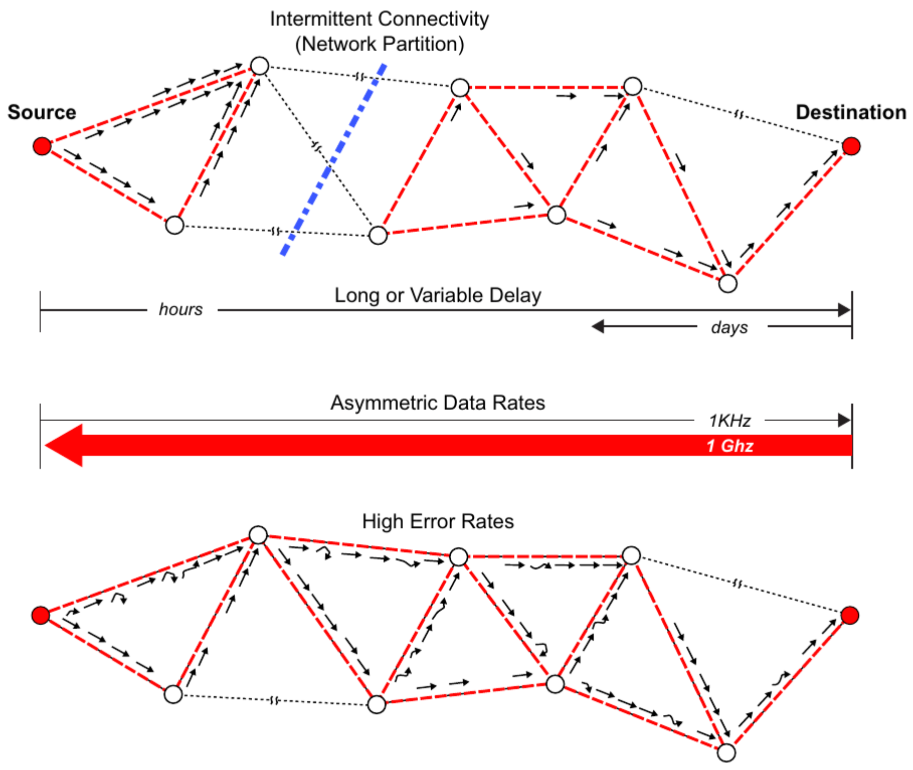

Figure 1.1: Illustration of the characteristics of challenged networks

1.1 DTN architecture

Delay-Tolerant Networking (DTN) was born in 2002 when Kevin Fall and other researchers, while working on the *Inter-Planetary* (IPN) project at the *Jet Propulsion Laboratory* (JPL) from NASA, decided to try and apply the basic concept of interplanetary communication to other terrestrial situations that had to face disruption and long delays. In the following years, more effort was punt into exploring this architecture until RFC 4838 was published in 2007 [RFC4838]. Citing this RFC:

> The DTN architecture addresses many of the problems of heterogeneous networks that must operate in environments subjected to long delays and discontinuous end-to-end connectivity. Is is based on asynchronous messaging and uses postal mail as a model of service classes and delivery semantics. It accommodates many different forms of connectivity, including scheduled, predicted, and opportunistically connected delivery paths. It introduces a novel approach to end-to-end reliability across frequently partitioned and unreliable networks. It also proposes a model for securing the network infrastructure against unauthorized access.

To overcome the limits set by TCP/IP, a new layer is added in the protocol stack, called *Bundle Layer*, typically between the Transport and Application layers. Inside of this layer is defined the *Bundle Protocol* (BP), whose operation is explained in the RFC 5050 (version 6) [RFC5050] and in the more recent RFC 9171 (version 7) [RFC9171].

A DTN node is defined as an entity that possesses a BP agent on top of a communication protocol at an inferior level (usually Transport).

The Bundle Protocol employs the so-called *Store-and-Forward* method. In order to handle disruptions between a DTN node and the next, each node first saves in a local database the bundle (*store*), even for extended periods of time (minutes or even hours), waiting for the possibility to send it (*forward*) to the next node, ultimately following a path that will reach the destination.

To better understand the DTN architecture, refer to Figure 1.2 which could, for instance, represent an interplanetary connection between Earth and Mars.

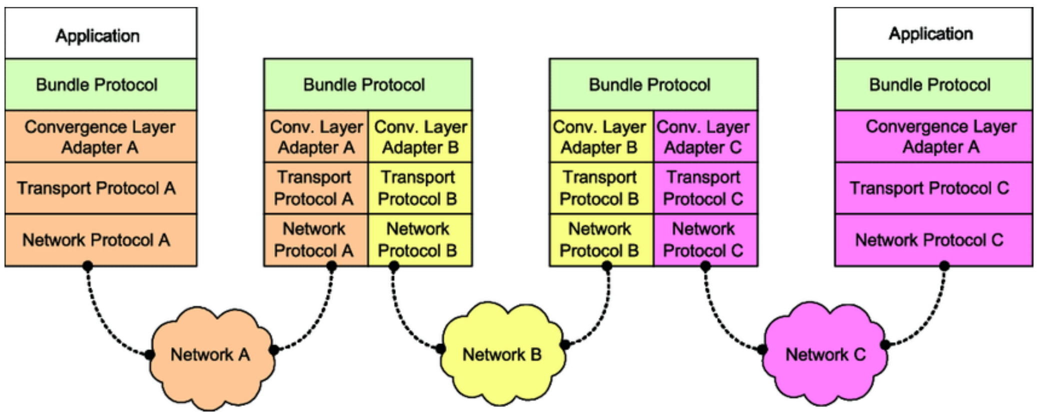

Figure 1.2: Example of DTN usage in heterogeneous environment

Let the entire path be divided into three segments, i.e. into 3 DTN hopes: the first segment between the source and a gateway located on Earth (with network A in between the two

DTN nodes), the second segment between the two gateways (with network B), and the third segment between the Martian gateway and the destination (with network C). Networks A, B, and C do not contain DTN nodes, but may contain many ordinary TCP-IP nodes, which are simply transparent to the DTN architecture. The two DTN nodes of the first segment, as well as those of the third, can use TCP or UDP below the Bundle Protocol, as there are no challenges. However, the same cannot be said for the interplanetary segment: in this case, a protocol suited for interplanetary connections, such as the *Licklider Transport Protocol* (LTP), is required. In order for the bundle protocol to interact with the underlying levels, an interface is needed. This interface is called *Convergence Layer Adapter*, and there is a specific version of it for each protocol that is intended to be used underneath the Bundle Protocol (e.g. TCPCL for TCP, LTPCL for LTP). This system allows the Bundle Protocol to send bundles on a vast variety of networks, without having to interact directly with the peculiarities of the different transport protocols. For example, the Bundle Protocol will not handle the TCP socket, but the TCPCL will.

Every DTN node is recognized by an *Endpoint Identifier* (EID). From the semantic point of view, this is an *Uniform Resource Identifier* (URI), that is, a string of characters that follows the scheme:

```
<scheme>:<scheme-specific-part>
```

where the first part represents the name of the scheme to be used (generally *dtn* or *ipn*), while the second part is a string whose meaning changes depending on the previous one. These two schemers follow different syntax:

- The *dtn* scheme, which was originally designed for terrestrial and interplanetary networks, follow the syntax
  ```
  dtn://machineID/appID
  ```
  where the first part (scheme and `machineID`) identifies the endpoint, while the final one (`appID`) represents a *demux token* which specify which is the application that generated the message port which it is to be delivered.
- The *ipn* scheme, which is mainly used by the Bundle Protocol to reduce overhead, utilizes the following syntax
  ```
  ipn:node_number.service_number
  ```

where the role of the two parts (`node_number` and `service_number`) are the same as the previous one, with the difference that they are numbers.

1.2 Hybla

The TCP protocol has been modified many times since it was first introduced. The sequence of RFC numbers cited in the recent RFC 9293 [RFC9293] is most likely the best proof of that. At the core of TCP there is the congestion control algorithm, and thus it is quite obvious that most modifications have interested this TCP component. The original algorithm and all most important "classic" variants (Tahoe, Reno, NewReno) consider losses as an indicator of congestion and rely on short RTT to follow congestion dynamics. Long RTTs impair TCP performance for the simple reason that the TCP congestion control is feedback based (it relies on the timely reception of ACKs), and as every protocol of this type, it cannot work properly if the feedback time is long. In particular, TCP performance is severely impaired when a long RTT connection (e.g. a GEO satellite one) has to compete with short RTT connections (e.g. terrestrial) for the bandwidth of a bottleneck. In practice, the long RTT connection "starves" and almost all the bandwidth is taken by the connection with short RTT. This problem is known in literature as "RTT unfairness". But even in the absence of competing traffic, long RTT connections are severely impaired by losses due to flipped bits, i.e. by noise, instead of congestion. TCP was designed under the assumption that losses due to noise are negligible with respect to losses due to congestion, so that halving the congestion window (CWIN), i.e. reducing the transmission rate, is the correct response. The cause of this problem is independent of the RTT, but the consequences are not. In fact, the longer the RTT, the longer the time necessary to reopen the CWIN. To cope with RTT unfairness, but also with the slow reopening of the CWIN, a new TCP variant, called TCP Hybla, was presented in 2004 by C. Caini and R. Firrincieli [Caini_2004].

The main point of the TCP Hybla proposal is to reduce the dependence of the congestion control on the round trip time. It does so through a modification of the standard rules for the congestion window increase, both in the slow start and in the congestion avoidance phases. Moreover, it requires the use of the SACK option and the use of timestamps. Further advantages are also obtained by implementing packet spacing techniques, which reduce the probability of buffer overflow at intermediate routers.

In most versions of TCP, when a connection is established, the sender probes for bandwidth availability by increasing the congestion window value W. In the *slow start* phase, starting from an initial value W_0, the congestion window is increased buy the number of bytes equal to the maximum segment size (MSS) per non-duplicate ACK received. It continues to grow this way until it reaches the slow start threshold (`ssthresh`). At this point, the source switches to the *congestion avoidance* (CA) phase, during which the congestion window is increased by MSS/W bytes per new ACK received. This rise continues until either the size of the receiver buffer (advertised window) is reached, or a segment is lost. Recovery actions are then taken depending on the specific TCP version. What differentiates Hybla is the introduction of a factor ρ defined as

$$\rho = RTT/RTT_0$$

where RTT_0 represents the round-trip time of the reference connection, which serves as the target for performance equalization. This factor is then used for the calculations of the CWIN, both in the slow start and in the congestion avoidance phases. Figure 1.3 shows the comparison between TCP standard and TCP Hybla for different RTT values. This graph shows how, while the transmission of data in TCP standard decays with the rise of RTT, in TCP Hybla the connections present the same performance as the one taken as reference.

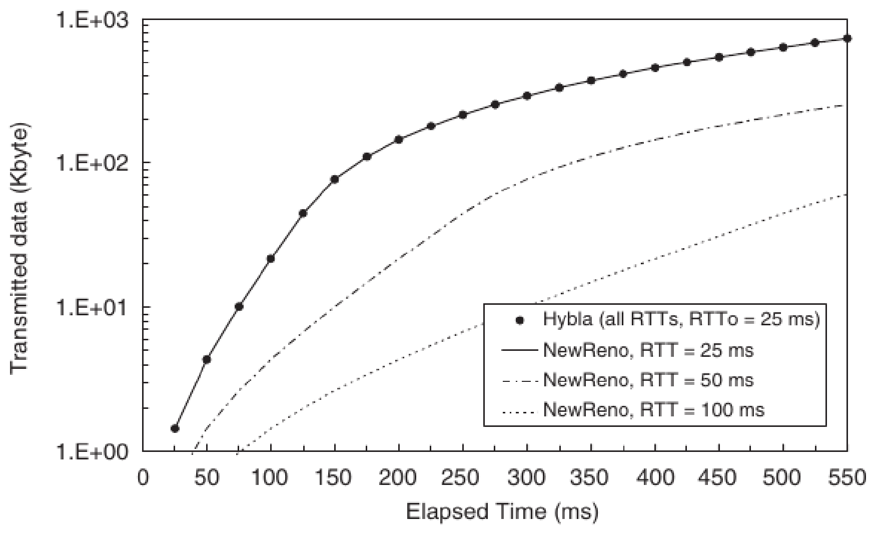

Figure 1.3: TCP Hybla vs. TCP standard

1.3 QUIC

QUIC is a general-purpose, connection-oriented, transport protocol initially designed at Google. Initially implemented and deployed in 2012, it was publicly announced in 2013, and in May 2021, it was officially standardized in the RFC 9000 [RFC9000] within the IETF. This protocol provides structured communication through flow-controlled streams, a reduced overhead to establish a connection, and native support for network path migration. QUIC also provides security measures employed to ensure confidentiality, integrity, and availability that can be taken advantage of in a wide variety of use cases.

One of the novelties introduced by QUIC is that it combines negotiation of cryptographic and transport parameters during its handshake. It does so by integrating the TLS handshake, although using a customized framing for protecting packets. A more detailed digression about this can be found in the dedicated RFC 9001 [RFC9001]. To summarize, the handshake is structured in a way that allows the exchange of application data as soon as possible. This includes an option for clients to send data immediately (0-RTT), which requires some sort of prior communication or configuration to be enabled.

Endpoints communicate in QUIC by exchanging QUIC packets. Most of these packets contain frames, whose task is to carry control information and application data between endpoints. Packets are carried in UDP datagrams to better facilitate deployment in existing systems and networks; QUIC will authenticate the entirety of each packet and encrypt as much as it is practical to do. QUIC also provides the necessary feedback to implement reliable delivery and congestion control. Another important addition of this protocol is the fact that connections are not strictly bound to a single network path: the migration uses connection identifiers to allow the transfer to a new network path. This design allows endpoints to continue the connection even after changes in the network topology or address mappings, such as those that could be caused by NAT re-binding. Applications can terminate the connection in multiple ways; for example, they could manage a graceful shutdown, negotiate a timeout period, or errors could cause immediate connection tear-down.

The basic mechanism for applications to exchange information over a QUIC connection is via streams, which are ordered sequences of bytes. Streams can be of two types: bidirectional, which allow both endpoints to send data, and unidirectional, which allow a single endpoint to send data on it. Streams are designed to be as less impactful as they can be on applications; for instance, a single STREAM frame can open, carry data for, and close a stream. Streams are identified within a connection by a numeric value, referred to as the stream ID. This number is a 62-bit integer (0 to $2^{62} - 1$) that is unique for all streams on a connection, so an endpoint must not reuse a stream ID within a connection. QUIC does not provide any means of ensuring ordering between bytes on different streams. On the other hand, data delivered on a stream must be an ordered byte stream; this requires the endpoints to buffer any data that is received out of order. Moreover, every endpoint must ensure that the data it intends to send is within the flow control limits set by its peer.

Applications can carry out different operations on streams depending on if they are on the sending or receiving part of a stream. On the sending part of a stream, an application protocol can [RFC9000]:

- **Write data**, understanding when stream flow control credit has successfully reserved to send the written data.
- **End the stream** (clean termination), resulting in a `STREAM` frame with the FIN bit set.
- **Reset the stream** (abrupt termination), resulting in a `RESET_STREAM` frame if the stream was not already in a terminal state.

On the receiving part of a stream, an application protocol can:

- **Read data**
- **Abort reading** of the stream and request closure, possibly resulting in a `STOP_SENDING` frame.

Also, an application could request to be informed of state changes on streams, including events like the opening and reset of a stream, when the peer has aborted the reading of a stream, when new data is available, and when data can or cannot be written to the stream due to flow control.

In RFC 9000 two state machines are presented: one for the sending part and another for the receiving part of a stream. These state machines are not compulsory, but serve as a baseline for all implementations: each implementation can decide to create its own states, but they must all be consistent with what is described in the RFC. Unidirectional streams use either one of the machines, depending on the type of stream and endpoint role. Bidirectional streams, on the other hand, use both states at both endpoints. The behavior is for the most part the same beyond the stream type.

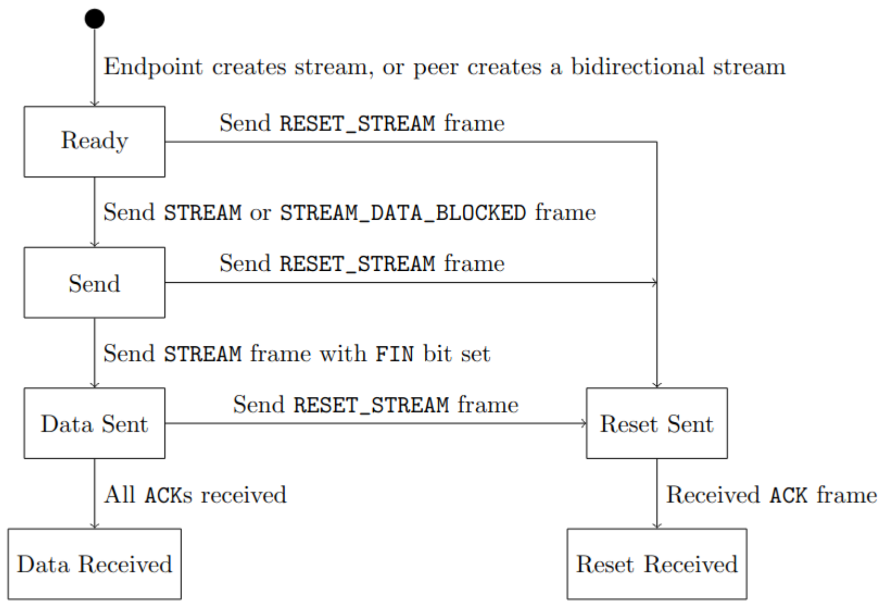

Figure 1.4: State machine representing the sending part of a stream

The sending stream states can be seen in Figure 1.4 When an application decides to initiate the communication, the sending part of the stream is opened. The "Ready" state is the first point in the lifetime of a stream and represents a newly created stream that is able to accept data from the application. Data can be buffered in this state in preparation for shipping. When the first `STREAM` or `STREAM_DATA_BLOCKED` frame is sent, the sending part of a stream enters the "Send" state. In this state, an endpoint transmits (and retransmits as necessary) stream data in `STREAM` frames, while always respecting the flow control limits imposed by its peer. For this purpose, the endpoint accepts and processes the `MAX_STREAM_DATA` frames, sent by the peer, that contain the maximum amount of data that can be sent on a stream. When the application has sent all stream data, it fires a frame containing the `FIN` bit and enters the "Data Sent" state. From here, the endpoint retransmits only stream data as necessary, and it can do so without checking the flow control limits. When all stream data has been successfully acknowledged, the sending part of the stream enters the "Data Recvd" state, which is a terminal state.

From any state that is one of "Ready", "Send" or "Data Sent", an application can signal that it wishes to abandon the transmission of stream data. On the other hand, an endpoint might receive a `STOP_SENDING` frame from its peer. In both cases, the endpoint sends a `RESET_STREAM` frame, which causes the stream to enter the "Reset Sent" state. Once this packet has been acknowledged, the sending part enters the "Reset Recvd" state, which is a terminal state.

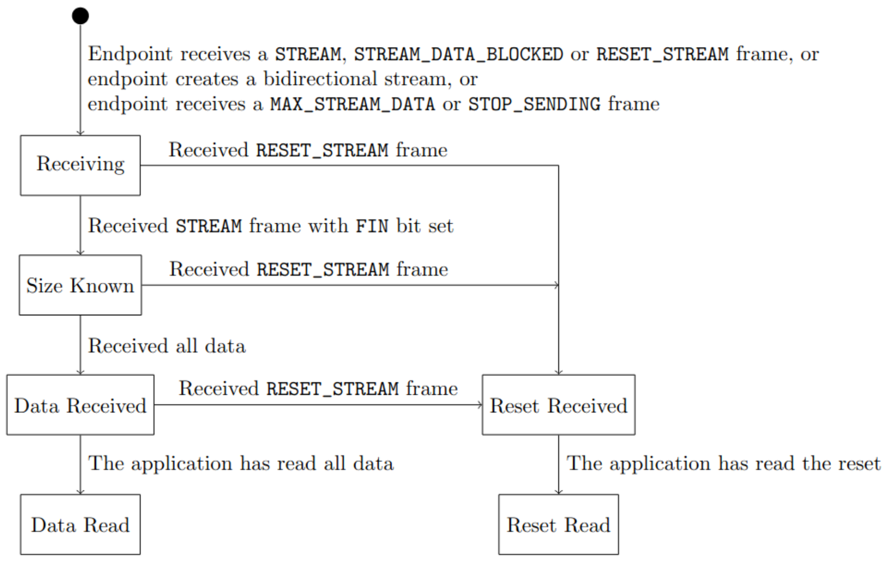
Endpoint receives a STREAM, STREAM_DATA_BLOCKED or RESET_STREAM frame, or
endpoint creates a bidirectional stream, or
endpoint receives a MAX_STREAM_DATA or STOP_SENDING frame
Receiving
Received RESET_STREAM frame
Received STREAM frame with FIN bit set
Size Known
Received RESET_STREAM frame
Received all data
Data Received
Received RESET_STREAM frame
Reset Received
The application has read all data
The application has read the reset
Data Read
Reset Read

Figure 1.5: State machine representing the receiving part of a stream

The receiving stream states can be seen in Figure 1.5. The states for receiving mirror only a part of the sending states: the receiving part does not track states on the sending machine that cannot be observed, such as the "Ready" state. Instead, it tracks the delivery of data to the application, some of which cannot be observed by the sender.

The receiving part of a stream is created when the first `STREAM`, `STREAM_DATA_BLOCKED`, or `RESET_STREAM` frame is received for a specific stream. In the case of a bidirectional stream, the receipt of a `MAX_STREAM_DATA` or `STOP_SENDING` frame for the sending part also creates the receiving part of a stream. In the bidirectional case, receiving a `MAX_STREAM_DATA` frame for an unopened stream signals that the peer has opened the stream and is providing flow control credit. Receiving a `STOP_SENDING` frame, instead, indicates that the peer no longer wishes to receive data on this stream. The initial state for receiving is "Recvd". In this state, the endpoint receives `STREAM` or `STREAM_DATA_BLOCKED` frames; incoming data is buffered and reassembled, if needed, before delivery to the application. As the application consumes data and the buffer becomes available, the endpoint sends `MAX_STREAM_DATA` frames to allow the peer to send more data. When a `STREAM` frame with a `FIN` bit is received, the final size of the stream is known and the machine enters the "Size Known" state. At this point, the endpoint only receives retransmissions of stream data without sending any more `MAX_STREAM_DATA` frames. When all the data has been received, the receiving part enters the "Data Recvd" state; this could happen as a result of receiving the same frame that caused the transition to "Size Known". This state persists until all data has been delivered to the application and when this happens the stream enters the "Data Read" state, which is a terminal state.

When in the "Recv" or "Size Known" state and the machine receives a `RESET_STREAM` frame, it will cause a transition to the "Reset Recvd" state that might stop the delivery of stream data to the application. This reception might also happen while in the "Data Recvd" state, meaning that all stream data have already been received. Similarly, it is possible for remaining stream data to arrive after receiving a `RESET_STREAM` frame, which is while being in the "Reset Recvd" state. An implementation is free to choose how to handle these situations. Once the application confirms the receipt of the reset signal, the receiving part transitions to the "Reset Read" state, which is a terminal state.

A QUIC connection is shared between a client and a server and begins with a handshake phase during which the two points establish a shared secret using the cryptographic handshake protocol and negotiate the application protocol. The handshake also is needed to confirm that both endpoints are willing to communicate and establishes the basic parameters for the connection. There is also the possibility to use 0-RTT, which means that the client will send application data to a server before receiving a response from it. However, using 0-RTT does not provide any protection against replay attacks from a malicious third party. A server can also send application data to a client before it receives the final cryptographic handshake messages that allow it to confirm the identity and liveness of the client.

Every connection has a set of identifiers that describe the connection itself. These numbers are independently selected by the endpoints, and each endpoint selects the connection IDs of its peer. The most important task of the connection IDs is to ensure that changes of addresses at lower layers (UDP, IP) do not interfere with the ability of a QUIC connection to deliver packets to the right endpoint. Multiple connection IDs are used in order to create confusion for an observer, since it would not be able to identify if the packets are being used for the same connection. It is also important, for this purpose, that connection IDs do not contain any type of information that could help the observer to correlate them.

1.3.1 Congestion control

Every QUIC packet is sent with a packet-level header that indicates the encryption level and also includes a *packet sequence number*. These numbers are never repeated during the lifetime of a connection, to prevent ambiguity, and it is permitted for some packet numbers to never be used, leaving intentional gaps.

QUIC packets can contain multiple frames of different types, and the recovery mechanisms ensure that those frames that need reliable delivery are acknowledged or declared lost and sent in new packets as necessary.

This design obviates the need for disambiguating between transmissions and retransmissions, removing some complexity from QUIC's interpretation of TCP loss detection mechanisms. In particular, QUIC separates transmission order from delivery order: packet numbers indicate transmission order, and delivery order is determined by the stream offsets in `STREAM` frames. Since the packet number is strictly increasing, a higher packet number signifies that the packet was sent later than a lower one. If a packet is detected lost, QUIC includes the required frames in a new packet, and it will give it a new packet number, removing ambiguity about which packet is acknowledged when an ACK is received. The same cannot be done in TCP, where most mechanisms implicitly try to infer the transmission ordering by checking the TCP sequence numbers. In addition to this, QUIC supports many ACK ranges, as opposed to TCP's three SACK ranges, speeding up recovery and reducing spurious retransmits.

RFC 9002 [RFC9002] specifies a sender-side congestion controller for QUIC similar to TCP NewReno. The signals provided for congestion control in this RFC are generic and designed to support different sender-side algorithms; this also allows each endpoint to unilaterally choose a different algorithm to use. A more detailed explanation of this mechanism can be found in RFC 9002.

1.4 QUIC implementations

Since its release, QUIC has been implemented in various different ways by several organizations, to which most of them proposed a different implementation of the protocol. As of now, this protocol is supported by all the major browsers. In 2012, it was experimentally developed in Google Chrome and was announced as part of Chromium version 29 (released in August 2013) [Chrome]. As of now, it is enabled by default in Chromium and Chrome, and it is used by more than half of all connections to Google's servers. It was released in Firefox starting from May 2021 [Firefox], and it has been officially supported in Safari since version 14 in 2020 as an optional feature [Safari24], later enabled by default in version 16.

As previously stated, there are currently several actively maintained implementations of the QUIC protocol. gQUIC is the original version of QUIC proposed by Google: it is this version that inspired the IETF to define a standard for this protocol.

The following is not meant to be an exhaustive list of every implementation, but rather a summary of the most known and used among them.

- Chromium has a specific version of QUIC that evolved from gQUIC (which is still being used by Google), and consequently used in Google Chrome. It is written in C++ and released under the BSD-3-Clause License. It can be found in all Chromium based browser, such as Google Chrome, Microsoft Edge or Brave, and also in some other Google services, like Gmail or YouTube. Since it is so widely used, this implementation is meticulously scrutinized, ensuring stability and efficiency. The inside mechanisms, such as congestion control, packet loss recovery, and low latency, are highly optimized and regularly updated to keep up with the latest QUIC and HTTP/3 improvements proposed by the IETF. The only downside is that this implementation is tightly integrated inside Chromium's networking stack, making it hard to use separately. For the same reason, the codebase is really large and optimized for Google's own infrastructure, and the documentation is mainly maintained for internal use.
- **quiche** is an high-performance QUIC and HTTP/3 implementation used in Cloudflare infrastructure, reverse proxies, clients, and general QUIC applications, released under the BSD-2-Clause License. It does not have memory-related vulnerabilities common in

C/C++ because it is written in Rust. It is actively maintained by Cloudfare and deployed in real-world services, since it can be embedded in various applications, and it is fully compatible with QUIC-based HTTP/3 communication. The disadvantages of this implementation is that it requires a Rust development environment, which might not be ideal for all users, and it is not widely adopted outside Cloudflare's ecosystem. It also requires external logic or integrations for advanced congestion algorithms.

- **MsQuic** is a cross-platform QUIC implementation, with great performance, optimized for Windows, Azure, and Microsoft services that can also work on Linux and macOS systems. Released under the MIT License and written in C, it is designed for low latency and high throughput applications. It is built to handle a large amount of concurrent connections efficiently using multi-threading, and it is well-documented and open-source: Microsoft provides detailed documentation and active support. The main drawback of this implementation is that, even though it works on different platforms, it is mainly optimized for Windows environments, and its API is more low-level than some other QUIC implementations, requiring more effort to integrate it.
- picoquic is a lightweight and extensible QUIC implementation, primarily used for research, experimentation, and custom protocol development. It is written in C and released under the MIT License. It is designed to minimize the amount of features it provides to make it as small and modular as possible. With its minimalist design, it allows easy modifications for new transport protocols and congestion control algorithms. It follows closely the latest standards dictated by the IETF QUIC specifications; it is open source and actively maintained by a community of contributors. It is not optimized for large-scale production, missing some performance tuning found, for example, in msQuic or quiche. It has limited built-in features, forcing developers to manually implement some advanced optimizations and congestion control mechanisms. It is also single-threaded, meaning it lacks the efficiency in high-performance use cases that other multi-threaded implementations, e.g. msQuic, could offer.

For the purposes of this thesis, picoquic was chosen for its minimalist design and the possibility to easily add new congestion controls. It also aligns with the standardization done by the IETF, so the implementation follows what is written in RFC 9000.

1.5 Aim of this thesis

The objective of this thesis is to develop the tools necessary for evaluating QUIC protocol performance in satellite and DTN networks, to carry tests, and then examine the results obtained. Concerning the tools, an already existing evaluation software (called picoquicdemo [Picoquicdemo]) has been augmented in order to enable time based experiments; the process of launching tests and collecting data in graphs has been completely automated by the combined use of a bash and a Python script. All tests were executed on a testbed of virtual machines provided by the virtualization software Virtualbricks [Virtualbricks].

This thesis is part of a broader research project that includes two companion theses, also developed at the German Aerospace Center (DLR), by fellow students Mattia Moffa [Moffa_2025] and Valentino Cavallotti [Cavallotti_2025]. The project's objective is that of investigating the potential application of QUIC in satellite and interplanetary networks, whether implemented as a Transport protocol or as a DTN convergence layer. In particular, Moffa's thesis consists in creating a QUIC Convergence Layer Adapter for Unibo-BP [Unibo-BP], the implementation of the Bundle Protocol developed by the University of Bologna. Cavallotti's thesis consists in the implementation of TCP's Hybla congestion control algorithm within Picoquic.

The present thesis is organized as follows:

- Chapter 2 gives an overview of the picoquic implementation and then follows up with a description of the testing software already implemented;
- Chapter 3 describes the additions that were made to picoquicdemo, how they are implemented and how to perform tests using the new features;
- Chapter 4 describe the tests that were done to measure the performance of QUIC used as a Transport protocol in satellite communications, and then presents the results obtained;
- Chapter 5 describes the tests that were done in order to evaluate the performance of QUICCL in a satellite DTN network, and then presents the results obtained;
- Chapter 6 draws the conclusions.

Chapter 2

Picoquic

At project start, it was necessary to choose the QUIC implementation to use in all three thesis; the choice was in favor of picoquic [Picoquic] because it was the most suitable for this project's aims. Firstly, the creator, Christian Huitema [Huitema] , wanted to create a lightweight QUIC implementation with the purpose of testing the standardization required by the IETF, thus picoquic is constantly updated to follow the latest releases of the standard. Secondly, the environment used for developing and testing these project is on a Linux machine, so picoquic aligned perfectly with the requirements of both machines and developers. Lastly, it was already known and used inside DLR.

This chapter will summarize the functioning of picoquic and its features; to know more, the reader is referred to Huitema's blog [Huitema] or the official GitHub page [Picoquic].

2.1 Overview

The main component of picoquic is the core library, which interfaces with applications through the public application API and with the system through the network API.

The main components of picoquic are the QUIC context and the QUIC connections. The former provides the data common to all connections and holds the tables necessary to route packets to specific connections. Connections are created within a QUIC context and each one of them holds path contexts, enabling the management of connection migration to different addresses or simultaneous use of multiple paths if multi-path extensions are enabled. The connection context also keeps track of the streams opened by the application or by the peer through a list, and also all the structures required to enable transmission of data according to the QUIC protocol.

Regarding the security of transmission, QUIC requires the use of TLS 1.3 [RFC8446] to negotiate encryption keys and verify certificates or public keys during the handshake. Picoquic utilizes the TLS implementation provided by picotls [Picotls], which, in turn, uses cryptographic functions provided by openSSL [openSSL].

The *public application* API of picoquic is described in the header file `picoquic.h`. Here are contained the data types and the functions that allow the applications to create QUIC contexts and QUIC connections, to receive callbacks when new connections are created or when data or other events happen on a connection, to create data streams and push data, and to close connections.

The *networking* API allows processes to submit incoming packets to a QUIC context, and to poll a QUIC context for new packets to send. A typical picoquic process will be organized as follows:

1. Create a QUIC context
2. If running as client, create the client connection
3. Initialize the network, for example by opening sockets
4. Loop:
    - Check how long the QUIC context can wait until the next action, i.e. get a time `t` using the *polling* API
    - Wait until either the timer `t` expires or packets are ready to be processed
    - Process all the packets that have arrived and submit them through the *incoming* API
    - Poll the QUIC context through the *prepare* API and send packets if they are ready
    - If errors happen during these operations, report issues using the *error notify* API
5. Exit the loop when the client connections are finished, or on a server if the server process needs to close
6. Close the QUIC context

One of the most important parts listed above is the *prepare* API. This API allows a process to poll the QUIC context and learn whether packets are ready or not. The signature of the function responsible for this is the following:

```
int picoquic_prepare_next_packet_ex(picoquic_quic_t* quic,
    uint64_t current_time, uint8_t* send_buffer,
    size_t send_buffer_max, size_t* send_length,
    struct sockaddr_storage* p_addr_to,
    struct sockaddr_storage* p_addr_from, int* if_index,
    picoquic_connection_id_t* log_cid, picoquic_cnx_t** p_last_cnx,
    size_t* send_msg_size);
```

where the arguments are:

- `quic` is a pointer to the QUIC context.
- `current_time` is the time of the call.
- `send_buffer` is the memory that will be used for the contents of one or several packets.
- `send_buffer_max` is the size of the send buffer.
- `send_length` is a pointer that will contain the amount of data written in the send buffer. It is set to zero if no packets are ready to be sent.
- `p_addr_to`, `p_addr_from` are pointers in which the IP addresses and ports to which and from which the packets shall be sent are written.
- `if_index` is the interface through which the packets shall be sent
- `log_cid` is an optional pointer: if not NULL, it will receive the value of a connection identifier that could be used to tag log messages relative to packets.
- `p_last_cnx` is an optional pointer: if not NULL, it will receive a pointer to the connection context for which the packets shall be sent, or NULL if the packets are sent outside of a connection context, such as for example stateless retry packets.

2.1.1 Packet loop

The *packet loop function* plays a key role in the event-driven process performed by picoquic; in particular, it runs an event loop that:

- **Receives packets** from the network.
- **Processes those packets** according to the QUIC protocol: handling acknowledgments,

retransmissions, congestion control, etc.

- **Sends packets** when necessary.
- **Handles timing events** like connection timeouts, scheduled retransmissions, and flow control updates.

This function will keep running until an external condition tells it to stop, such as a user request, an error, or a termination signal. It is used in both QUIC clients and servers to direct the flow of the communication.

During the execution of the packet loop, picoquic can decide to call two different functions: the packet loop callback and the stream data callback. The *packet loop callback* gets called at specific key points and allows an application to interact with the loop without changing its core behavior. The callback is triggered in different situations, such as:

- When the loop is **starting up**, useful for initializing resources.
- After a packet has been **received**, allowing applications to inspect or modify packet handling.
- After a packet has been **sent**, useful for tracking outgoing data.
- When the loop is **about to shutdown**, allowing for cleanup operations.

In picoquic, the definition of the states of this callback can be found inside the `picoquic_packet_loop.h` header file [Picoquic].

A list of all of them and a brief explanation can be seen in the following table:

Name	When it is called
`picoquic_packet_loop_ready`	The packet loop is operational.
`picoquic_packet_loop_after_receive`	The application received a specific number of packets.
`picoquic_packet_loop_after_send`	The application sent a specific number of packets.
`picoquic_packet_loop_port_update`	Used to communicate to the application that a new UDP socket was opened.

Name	**When it is called**
`picoquic_packet_loop_time_check`	(Optional) triggered at each iteration of the packet loop; it provides the current time and the time elapsed since last iteration.
`picoquic_packet_loop_system_call_duration`	(Optional) it provides the duration of the system call select, used by picoquic to check for events.
`picoquic_packet_loop_wake_up`	Used when *loop wakeup* is supported to trigger the packet loop for specific events.
`picoquic_packet_loop_alt_port`	Provides an alternative port for testing multi-path or migration.

The other important function in picoquic is the *stream data callback*. Its purpose is to handle incoming QUIC stream data, so it is triggered whenever data arrives on a stream. It informs the application about different events, such as:

- **New data arriving** on a specific stream.
- **End of a stream**, when the sender signals that it has finished sending (when the FIN is set).
- **Stream reset**, if an error or manual interruption occurs.
- **Connection closure**, when the entire QUIC connection is terminated.

The definition of the events for this callback can be found in the header file `picoquic.h` [Picoquic]. There are a lot of events that are not useful for the purpose of this thesis, so the following table will present only the one that are of interest:

Name	**When it is called**
`picoquic_callback_stream_data`	Data received from peer on a specific stream.
`picoquic_callback_stream_fin`	FIN received from peer on a specific stream; data is optional.

Name	When it is called
`picoquic_callback_prepare_to_send`	Ask the application to send data in frame.
`picoquic_callback_stream_reset`	Reset Stream received from peer on a specific stream.
`picoquic_callback_stateless_reset`	Stateless reset received from peer.
`picoquic_callback_stop_sending`	Stop sending received from peer on a specific stream.
`picoquic_callback_close`	Connection closed.
`picoquic_callback_application_close`	Application closed by peer.
`picoquic_callback_version_negotiation`	Version negotiation requested.
`picoquic_callback_stream_gap`	Gap indication in data provided on stream.
`picoquic_callback_almost_ready`	Data can be sent, but the connection is not fully established.
`picoquic_callback_ready`	Data can be sent and received, connection migration can be initiated.
`picoquic_callback_request_alpn_list`	Provide the list of supported ALPN, explained in the following pages [RFC7301].
`picoquic_callback_set_alpn`	Set ALPN to negotiated value.

2.1.2 Sending and receiving data

Picoquic provides two APIs for sending data on stream: a queuing API and a *just in time* API. The receiving portion is always just in time, using the callback provided by the application. As stated previously, streams can belong to one of four classes: bidirectional or, opened by the client or by the server. The stream identifier is a 64 bit number where the least 2 significant bits represent the type of the stream. In picoquic, streams are created when the application starts using them or when the first data arrive from the peer.

The difference between the queuing and just in time approach is that the latter does not create an internal data queue. The application is instead called when there is the possibility to send a stream frame. It then gets access to the packet that is being formatted, and writes the data directly into it just before it is sent on the network. Using this approach is slightly more complex, but it can reduce the consumption of memory and CPU usage.

When using the just in time API, the application:

- Marks a stream as *active* using the function `picoquic_mark_active_stream`.
- Receives the callback `picoquic_callback_prepare_to_send` when picoquic is ready to send data on a stream.
- Services the callback by reserving a buffer using the function `picoquic_provide_stream_data_buffer` and then copying into it the application data.

The functions just described are documented inside the `picoquic.h` header file [Picoquic], and have the following signature:

```
int picoquic_mark_active_stream(picoquic_cnx_t* cnx,
    uint64_t stream_id, int is_active, void* v_stream_ctx);

uint8_t* picoquic_provide_stream_data_buffer(void* context,
    size_t nb_bytes, int is_fin, int is_still_active);
```

When a stream is active, it will be polled for data **only after** all currently queued data has been sent, meaning that picoquic always prioritizes retransmissions before sending new data. Once the application gets the callback to prepare data to send, it also receives an argument indicating the largest amount of data that can be sent. The function that is used to provide the buffer receives in input the number of bytes that it wants to write, an indication of whether or not the FIN of the stream was reached, and also an indication of whether or not the stream is still active. Usually, these two numbers are opposites: if `is_fin` is 0 and `is_active` is 1, then it means that the application still has data to send, while if `is_fin` is 1 and `is_active` is 0, then it means that the application does not have any more data to send. There is also the possibility of those two to be 0 at the same time, meaning that the application has finished sending all its data, but in the future it might want to send again on that stream. Obviously, the application will have to call the `picoquic_mark_active_stream` function later and mark the stream as active again.

When stream data are received, the contents are queued temporarily until it can be delivered in order. At this point, the application will receive the callback `picoquic_callback_stream_` `data` or, if the packet arrived contains a FIN, `picoquic_callback_stream_fin`. After this, the application is free to digest the data as it see fits.

The procedure of closing a stream is trivial. When an application has finished sending data on a stream, it will set the FIN bit to 1. When the peer receives the last frame (in order), the application receives the `picoquic_callback_stream_fin callback`. It is not necessary to send a FIN alone, it can also be aggregated with stream data.

Applications can also abruptly close the stream using the reset mechanism. They can do so by calling the function:

```
int picoquic_reset_stream(picoquic_cnx_t* cnx,
    uint64_t stream_id, uint64_t local_stream_error);
```

The peer's application will receive the callback `picoquic_callback_stream_reset`. This mechanism only works for the sending direction of a stream, meaning that, if a stream is bidirectional, the peer might still send data on their side until they decide to close or reset it. If an application resets a stream, picoquic will immediately stop sending any type of data on that stream, including retransmissions needed because of packet loss.

Bidirectional streams are considered closed by picoquic if they are closed or reset in both directions, meaning that both sides have sent a FIN or a reset and received the corresponding callback. Unidirectional streams are considered closed when they send a FIN, if the application is the sender, or when the callback is received, if the application is on the receiver side. After this, all resources associated with the stream are freed.

2.2 Picoquicdemo

Picoquic offers a suite of programs that can be used for different purposes. One of these is **picoquicdemo**, created to assist developers in testing the new code. It is a client-server program based on the use of an application protocol specifically designed to test QUIC performance. Although at thesis start only quic-perf [quic-perf_draft] was available inside the picoquic project, picoquicdemo can specify the one to use by means of the Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension [RFC7301] (ALPN for short). Thanks to this, developing an extension of the original quic-perf, called picoquicperf (described in the next chapter), was possible. To select which ALPN to utilize, one can simply use the option `-a <ALPN>` when launching picoquicdemo.

2.2.1 Usage of picoquicdemo

Picoquicdemo server does not require any option; a default port is chosen (`4443`), but it can be changed by use of the `-p <number>` option, as shown below:

```
picoquicdemo -p 4444
```

Picoquicdemo client, vice versa, requires many parameters, with a syntax that is far from being user-friendly. First, the application protocol to be used for QUIC performance evaluation must be specified; then the address (IP and port) of the serve; lastly, the options specific to the protocol chosen, between quotation marks. An example is in order:

```
picoquicdemo -a perf 10.0.3.6 4444 "*1:0:-:1000:1000000;"
```

Note that quic-perf is denoted as “perf” in picoquicdemo. Options that are specific to quic-perf will be explained later. Some picoquicdemo options that are worth mentioning, valid for both server and client, are:

- `-G <cc_algorithm>`: use the specified congestion control algorithm. The possible choice are: NewReno [RFC6582], Cubic [RFC9438], BBR [BBR_draft], Fast [Fast_TCP] or Hybla, the last added by fellow student Cavallotti [Cavallotti_2025]. The default is BBR.
- `-l <file>`: text log file; if the file name is “`-`”the logging is sent to standard output. It only logs the first 100 packets. No text logging if absent.
- `-L`: log all packets, if absent the logging will stop after 100 packets.
- `-b <folder>`: the name of the directory where binary log files are saved; the name of the file will be the connection ID. No binary logging if absent.
- `-q <folder>`: the name of the directory where Qlog files are saved. Qlog is a specific type of logging conceived specifically for QUIC [Qlog_draft]. No qlog logging if absent. Another tool that can be used in conjunction with picoquicdemo is *picolog*: a small program that can generate qlog logs (or even a csv file) starting from the binary logging.
- `-F <file>`: append a raw report with summarized information about the performance achieved during the connection to a csv file. This report contains, among others, infor-

mation about the sending and receiving rate, the number of retransmissions, and the largest value the congestion window reached.

- `-h`: to show the help message.

It is also worth mentioning the option `-1`, which is server specific; using this option means that the server accepts only one connection. This was largely used in the testing portion of the thesis.

As a further example, the following command launches a picoquicdemo server that closes after one connection, whose port is 4444, and uses Cubic as congestion control on the reverse direction, i.e. from server to client:

```
picoquicdemo -p 4444 -G cubic -1
```

2.2.2 Usage of quic-perf

On the client side, if the application protocol used is quic-perf, one must also specify the test “scenario” to execute. In other words, at the end of the picoquicdemo command line there must be a quoted text that contains one or more scenarios, separated by a semicolon. All these scenarios are part of the same connection. Before presenting an example, it is important to stress the fact that there is no documentation and the quic-perf protocol has changed after the fork for the development of picoquicperf. For this reasons, the following explanation refers to the quic-perf version that was forked, and is best effort. It is also propaedeutic to the explanation of picoquicperf that will be given in the next chapter.

The example below refers to a single scenario, for simplicity:

```
picoquicdemo -G cubic -a perf -b ./logs_dir 10.0.3.6 4444 "*1:0:-1000:1000000;"
```

The scenario syntax, i.e. the meaning of the string field, is given below:

- The first number represents a repetition factor, representing the number of streams that will execute the same task. An asterisk must be placed before the number. In the example above is one.
- The second number represents the stream ID that will be used. Since in QUIC the last two bits of the stream ID are reserved to denote the stream type, two consecutive streams of the same type are at an interval of four, e.g. for a bidirectional stream opened by

the client the first ID is 0, the next are 4, 8, etc. This is of interest when consecutive scenarios are specified. It can be omitted, by placing a dash instead of a number; in this case picoquic will automatically use the default type and apply the logical order. In the example this is zero.

- The third number represents the previous stream ID, used in order to let the next scenario start when the first one ends. The number can be replaced by a dash, as in the example.
- The fourth number gives the bytes that the client must send to the server (forward direction); in the example is 1 kB
- The fifth number is the amount of bytes that the server must send to the client, in the example is 1 MB.

Chapter 3

Picoquicperf

Picoquicperf [Picoquic_MAC] is the fork of quic-perf that has been developed to make it more suitable to the planned performance evaluation. This protocol is selectable by using the ALPN `pico-perf`.

3.1 New features

3.1.1 Connection type

The main difference with the original quic-perf lies in the possibility to choose a time mode for the scenarios. Using the data oriented type of connection means that the client will first send all the data specified in the scenario to the server, which then sends back its portion of data. The aim of the tests is to measure the performance of the components, so it is far more useful to have a mode of functioning in which one part keeps sending at its maximum possibility for a specific period of time. This is why the time oriented approach was introduced in the first place. The picoquicperf syntax is inherited from that of quic-perf, with a few necessary modifications as shown below:

```
"<cnx_type>:*<repeat_count>:<stream_id>:<prev_stream_id>:<cnx_details>;"
```

The new fields are the first and the last:

- `<cnx_type>` can either be `D` for data oriented or `T` for time oriented approach.
- `<cnx_details>` differs based on the connection type:

  - For the data oriented mode, this part contains two numbers separated by a colon, indicating the number of bytes that the client will send to the server and vice versa.
  - For the time oriented mode, this part contains the wanted duration duration, an integer followed by s for seconds, ms for milliseconds, or us for microseconds.

As an example, a connection that uses the time mode has a scenario description that looks like this:

```
"T:*1:0:-:60s;"
```

which means that the client will send data for a total of 60 seconds.

An example of a data mode connection is the following scenario description:

```
"D:*1:0:-:100000:100000"
```

which means both the client and the server will send 100 kB of data.

As in quic-perf, inserting a number of repeat count grater than one means that picoquicdemo will execute that test using multiple streams. It will automatically generate more streams with increasing number according to default QUIC ordering. It is also possible to execute more scenarios one after the other or even two (or more) at a time. In both cases, the multiple scenarios must be concatenated in the same string, separated by a semicolon. By indicating the ID of the previous stream as previous stream ID of the subsequent test, two scenarios are executed in series, while by inserting the same number as the previous stream ID they are executed in parallel. For instance, the scenario description:

```
"T:*1:0:-:60s;D:*1:4:0:100000:100000"
```

means that the second stream is opened only when the first one is closed. Meanwhile, to have them execute at the same time, the scenario description must be:

```
"T:*1:0:-:60s;D:*1:4:-:100000:100000"
```

The difference is that, instead of a `0` in the previous stream ID field of the second experiment, a dash is present.

Using this syntax, it is possible to execute complicated scenarios, if one wishes to do so. In the case in which one test of a scenario has a repeat count greater than one, it is important to keep in mind that the picoquicperf protocol will assign increasing number of streams automatically. This means that when launching a test that has, for example, a 2 as repeat count, the following test in the scenario must have a stream ID of 8, because the ID 0 and 4 are already used in the previous test.

3.1.2 Logging

Another extension concerns real time logs printed on standard output. There is not an option to toggle this feature, but instead it is always active, since adding another option would add more complexity to the syntax, and in general this feature is incredibly useful. The quicperf protocol does not show anything on standard output during test execution, and the only feedback for the user consists in a summary report provided by picoquicdemo at client side, when the test end. This lack of information during test execution proved to be a significant limitation, as it was not possible to verify the regular evolution of tests.

For this reason, we added in picoquicperf a log function that prints a line on standard output after a certain time from the last log has elapsed (the current value is 5s, but is can be easily changed by means of a define). Logs can also be printed on a .csv file whose name is set by means of an environment variable, to be inserted before the command, as shown in the example below:

```
PICOQUICPERF_CSV_LOG_NAME=mytest.csv picoquicdemo -p4444
```

Logs are related to events, as the time elapsed since the last log is checked only after specific events, such as the reception of new data, happen. Therefore, the interval between logs is destined to be equal or greater than the wanted value. For example, if the event occurs after 4.9 seconds from the last printing, nothing is shown; if the next event triggers a check after 6 seconds, then it is printed. In the latter case the interval will result of 6 seconds instead of 5. A FIN will always trigger a print, no matter the time elapsed. The next line is an example of a log:

```
** [   5s ] - SID   0: Data passed to picoquic (B): 0 - rcvd (B): 56568417 --- Tx rate
↪  (Mbit/s), UP: 0.000 - DOWN: 90.509 **
```

The first number represents the time elapsed since the beginning of the test. The SID represents the stream ID to which that line of logging refers (let us anticipate that there are lines referring to single streams and other to the aggregate traffic). The next two numbers are the amount of bytes that picoquicdemo has passed to QUIC and the amount of data that has been received from it (ordered). The final two fields report the transmission and download rates in megabit per second, averaged on the time elapsed since the start.

There are a few type of logging lines, in different colors:

- **White**: the one just explained.
- **Yellow**: this is visible only when using multiple streams at the same time, and it will show the aggregated statistics of all the streams. A yellow line is shown after the white lines of the streams.
- **Purple**: this line is printed as soon as a stream has finished its lifetime, regardless of time elapsed from the last print.
- **Green**: this line is print at the end of the experiment and reports the summary of the connection: total data sent and received, average transmission rate and download rate over the whole duration. Like the purple one, this line is printed immediately.

When dealing with multiple scenarios that are executed one after the other, lines that refer to a single stream, i.e. in every color except green, have all fields reset. This means that every different test inside the scenario will have its own report. The green one will still show everything that has happened during the connection.

3.2 Implementation

This section aims to describe the most important parts of the code of picoquicdemo and picoquicperf. For more information, see the source code at [Picoquic_MAC].

The code added inside the `picoquicdemo.c` file is mainly used to set up every component that is necessary for the correct functioning of picoquic.

The configuration of the contest when picoquicperf is chosen as application protocol is the following:

```c
if (config->alpn != NULL && strcmp(config->alpn, UNIBO_QUICPERF_ALPN) == 0) {
    /* Set up a PICOQUICPERF client*/
    is_picoquicperf = 1;

    picoquicperf_ctx = picoquicperf_create_ctx(client_scenario_text);

    if (picoquicperf_ctx == NULL) {
        fprintf(stdout, "Could not get ready to run UNIBO_QUICPERF\n");
        return -1;
    }

    fprintf(stdout, "Getting ready to run UNIBO_QUICPERF\n");
}
```

This snippet simply calls the function `picoquicperf_create_ctx` that is defined inside the `picoquicperf.c` file.

To set up the stream data callback, the function `picoquic_set_callback` is used.

Beside the logging provided by the colored lines, at the end of each connection the client will show various statistics. In particular, picoquicdemo, when using the picoquicperf protocol, will show lines about the duration, the number of streams that were opened, the transmitting and downloading rate, and the amount of data sent and received.

The heart of the protocol lies inside the `picoperf.c` file. At its beginning there are two definitions of constants:

```
#define LOGGING_TIME 5000000
#define WRITING_ON_CSV_TIME 500000
```

The former is the amount of microseconds that the logging system has to wait before printing a new log line on the standard output. The latter refers to the same kind of lines, but printed in a .csv file.

The next important section of the code is the one that parses the scenario provided by command line. In order to do so, the `picoquicperf_parse_stream_desc` is used. This function will then call other utilities to parse the corresponding value and place it inside the stream context. The function is structured as follows:

```c
char const* picoquicperf_parse_stream_desc(char const* text, uint64_t default_stream,
    uint64_t default_previous, picoquicperf_stream_desc_t* desc)
{
    text = picoquicperf_parse_cnx_type(picoquicperf_parse_stream_spaces(text), &desc->type);

    if (text != NULL) {
        text = picoquicperf_parse_stream_repeat(text, &desc->repeat_count);
    }

    if (text != NULL) {
        text = picoquicperf_parse_stream_number(text, default_stream, &desc->stream_id);
    }

    if (text != NULL) {
        text = picoquicperf_parse_stream_number(text, default_previous,
            &desc->previous_stream_id);
    }

    if (desc->type == picoquicperf_data_oriented) {
        if (text != NULL) {
            text = picoquicperf_parse_post_size(picoquicperf_parse_stream_spaces(text), 0,
                &desc->post_size);
        }

        if (text != NULL) {
            text = picoquicperf_parse_response_size(picoquicperf_parse_stream_spaces(text),
                &desc->is_infinite, &desc->response_size);
        }
    }
    else if (desc->type == picoquicperf_time_oriented) {
        if (text != NULL) {
            text = picoquicperf_parse_duration(picoquicperf_parse_stream_spaces(text), 0,
                &desc->duration);
        }
    }

    /* Skip the final ';' */
    if (text != NULL) {
        if (*text == ';') {
            text++;
        }
        else if (*text != 0) {
            text = NULL;
        }
    }

    return text;
}
```

This function is called inside the `picoquicperf_parse_scenario_desc`, indicating the default stream ID (`0`) and the default previous stream ID (`UINT64_MAX`).

When creating a context for picoquicperf inside the picoquicdemo.c file, what happens is that the picoquicperf function `picoquicperf_create_ctx` first parses the scenario description and then initialize a tree data structure in which every stream represents a branch. This tree is populated when calling the function `picoquicperf_init_streams_from_scenario`, on the client side the first time when receiving the `picoquic_callback_ready` callback. The next streams are initialized when the previous stream, or streams, are completely closed. It is important to know that the polling system of picoquic, when looking for active streams on which to send data, it utilizes the following system: first it checks for the stream that has the lowest priority number, and then chooses that one if it's the only one with that level of priority. If two or more streams share the same priority, picoquic serves the lowest stream ID number first, if the priority is an odd number, or use a round robin method if the stream ID number is even. In picoquicperf all streams are marked as active with a priority of 8 (9 is the default), since the round robin method is more suitable to the functioning of this program.

The next big part of the source code is the definition of the callback function. This function contains a switch that redirects the execution in the right direction based on the type of event that woke the callback. The callbacks `picoquic_callback_stream_data` and `picoquic_callback_stream_fin` trigger the execution of the `picoquicperf_process_stream_data`. This function, on client side:

- Updates the values for the statistics regarding the number of bytes received.
- In data mode, checks if the amount of data received exceeds the amount specified in the scenario; if true, signals to stop sending and prepare to close the stream.
- In time mode, checks if the stream lifetime exceeds the amount specified in the scenario; if true, signals to stop sending and prepare to close the stream.
- If the data received contains a FIN, then the preparation of the stream closure is initiated.
- At the end, if the stream was prepared to be closed, it triggers the initialization of the next stream, or streams, from the scenario that have the field of previous stream ID equal to the ID of the stream that was just closed. The context of this stream is deleted if there are more streams that can be opened, otherwise the scenarios are over, which means that the connection can be closed.

Meanwhile on server side:

- Creates the stream context if this is the first byte that has arrived on this stream.
- Since the scenario is written on the client's side, this one informs the server of how much data it has to send back by putting this amount in the first 8 bytes of the first packet sent. When receiving data for the first time on a stream, the server waits for all the bytes containing this information. This means that the client has to send at least 8 bytes, when in data mode. In time mode this problem doesn't exists, since it's only the client that sends data and it will put all 0 in those first 8 bytes).
- After this, the function updates the values for the statistics regarding the number of bytes received.
- If it receives a FIN, first it will check if the number of bytes received is less than 8; in this case it sets the response size to zero and the stream is closed. Otherwise, its side of the stream is set to active, meaning that the server can start sending the data (if it supposed to).

The callback `picoquic_callback_prepare_to_send` triggers the execution of the `picoquicperf_prepare_to_send` function. This function:

- First checks if it has to send a FIN; to do so, it checks, in data mode, if the amount of byte sent is already the maximum or, in time mode, if the time passed is greater than the timer set in the scenario.
- Then it requests a buffer using the `picoquic_provide_stream_data_buffer`. On the client's side, it then fills the first 8 bytes that will instruct the server on the amount of data to be sent back. After that, the buffer is filled with dummy text.
- Only the server, if the packet to send contains a FIN, the stream is closed at the end of the function.

The other callbacks are less complex:

- The `picoquic_callback_stream_reset` callback triggers the call to `picoquic_reset_stream`.
- The `picoquic_callback_stop_sending` callback sets a flag on the stream that causes the peer to stop sending on its side of the stream.

- The `picoquic_callback_stateless_reset`, `picoquic_callback_close`, and `picoquic_callback_application_close` callbacks trigger the deletion of the picoquicperf context on the server side and, for both sides, set the callback to NULL.
- The `picoquic_callback_almost_ready` callback only prints a status message on the standard output.
- The `picoquic_callback_ready` callback initiates the streams from the scenario by calling the `picoquicperf_init_streams_from_scenario` function.
- The other callbacks are not implemented since they are not relevant for the purposes of this small program.

Finally, the logging system is implemented by two short functions that are called every time an event is triggered and enough time as elapsed. For instance, these functions are: `picoquicperf_print_info` and `picoquicperf_print_aggr_info`. Their structure is the same, so only the former will be shown here:

```c
void unibo_quicperf_print_info(picoquic_cnx_t* cnx, unibo_quicperf_ctx_t* ctx,
        unibo_quicperf_stream_ctx_t* stream_ctx, uint64_t stream_id, int stream_done,
        uint64_t current_time, int* nb_printed_info)
{
    if (stream_ctx == NULL && !ctx->is_client) {
        stream_ctx = unibo_quicperf_find_stream_ctx(ctx, stream_id);
    }

    // uptime is in microseconds
    uint64_t uptime = current_time - stream_ctx->post_time;
    uint64_t uptime_sec = uptime / 1000000;
    uint64_t data_passed = (ctx->is_client) ?
        stream_ctx->nb_post_bytes : stream_ctx->nb_response_bytes;
    uint64_t data_rcvd = (ctx->is_client) ?
        stream_ctx->nb_response_bytes : stream_ctx->nb_post_bytes;
    double up_rate = (ctx->is_client) ?
        ((stream_ctx->nb_post_bytes * 8.0) / uptime) :
        ((stream_ctx->nb_response_bytes * 8.0) / uptime);

    double down_rate = (ctx->is_client) ?
        ((stream_ctx->nb_response_bytes * 8.0) / uptime) :
        ((stream_ctx->nb_post_bytes * 8.0) / uptime);

    if (!stream_done) {
        if (current_time - stream_ctx->last_printed_info_time >= LOGGING_TIME) {
            fprintf(stdout,
            "** [ %3lus ] - SID %3lu: Data passed to picoquic (B): %ld - rcvd (B): %ld --- Tx
            ↪ rate (Mbit/s), UP: %.3lf - DOWN: %.3lf **\n",
            uptime_sec, stream_id, data_passed, data_rcvd, up_rate, down_rate);
            fflush(stdout);
            stream_ctx->last_printed_info_time =
                picoquic_get_quic_time(picoquic_get_quic_ctx(cnx));
            (*nb_printed_info)++;
        }
    } else {
        // end of stream
        fprintf(stdout, "\e[0;95m** [ %3lus ] - SID %3lu: Data passed to picoquic (B): %ld -
        ↪  rcvd (B): %ld --- Tx rate (Mbit/s), UP: %.3lf - DOWN: %.3lf - FIN **\e[0m\n",
        uptime_sec, stream_id, data_passed, data_rcvd, up_rate, down_rate);
        fflush(stdout);

        if (stream_ctx->csv_file != NULL) {
            fprintf(stream_ctx->csv_file, "%lu,%lu,%ld,%ld,%lf,%lf\n",
                uptime, stream_id, data_passed, data_rcvd, up_rate, down_rate);
            fclose(stream_ctx->csv_file);
        }
    }
}
```

The function first computes all the various values and then prints them on the standard output, changing the format depending on whether or not the FIN flag is present.

This function also mentions another method: `picoquicperf_print_info_on_file`, which is used to print on a .csv file the same information that are printed on standard output. The reason for this is that these information are not easily obtainable from the logging systems proposed by picoquic, and, since they are important for the evaluation process, a custom function is used.

3.3 Brief picoquicperf user guide

For the following tests, the same server is used. To launch it, the command is:

```
picoquicdemo -p 4444 -F picoserver.csv
```

This command starts a server that listens on the port 4444 and writes a raw summary of every scenario inside the same .csv file.

The first client is launched with the following command:

```
picoquicdemo -n test -G hybla -a pico-perf 10.0.3.6 4444 "D:*1:0:-:100000:10000000;"
```

The client utilizes Hybla as congestion control, connect to the server at the IPv4 address

10.0.3.6, port 4444, and open one stream for the scenario. In particular, the client will send 100 kB and the server will send 10 MB. This is a very simple test but it is through tests like this that we were able to spot errors and bugs.

To showcase the use of the time oriented mode, the following command could be used:

```
picoquicdemo -n test -G hybla -a pico-perf 10.0.3.6 4444 "T:*1:0:-:60s;"
```

This will create a client with the same characteristic as the one before, with the only difference being that this will keep sending data for a minute, and then stop.

For some more complex test, one could run the following:

```
picoquicdemo -n test -G hybla -a pico-perf 10.0.3.6 4444 "T:*2:0:-:60s;D:*1:8:-:100000:100000"
```

which will create three streams, the first two will send (on the client's side) for 60 seconds, while the third will see the client and the server exchange 100 kB 100kB of data. All these streams start at the same time.

To run the test but with the third stream starting after the first two are complete, on could run the following command:

```
picoquicdemo -n test -G hybla -a pico-perf 10.0.3.6 4444 "T:*2:0:-:60s;D:*1:8:0:100000:100000"
```

The only difference is in the previous stream ID field of the second description. Technically, putting a 0 means that the third will start after the stream ID 0 is closed, but it should correspond to the end of the stream with ID 4 since they have the same duration.

Chapter 4

Performance evaluation of QUIC in a few selected satellite networks

For the purposes of this thesis, the cases that are examined are distinguished in two chapters. The current section is focused on the tests executed in order to evaluate QUIC, in particular using picoquic, as a peer-to-peer tool in satellite communications (using LEO and GEO satellites). The metrics that are presented are focused on the values of the congestion window and the goodput of the connections, with a particular focus on Hybla.

The next chapter contains the tests that are focused on the use of QUIC, in particular using QUICCL, for satellite communications that take place in a DTN network.

4.1 Testbed layout and tools

The testbed utilized for the tests is part of a project that can be downloaded from the CNRL (Challenged Network Research Lab) website [CNRL] and then opened in Virtualbricks [Virtualbricks]. This software can be used to easily create virtual machines and connect them through channel emulators: all the components can be customized for the specific needs, for example a channel emulator could add a delay in one or both ways, or also add losses and a limit to the bandwidth.

The virtual machines can be accessed using the ssh command, which also means that it is possible to launch a command from the host to a VM, as in the example below:

```
ssh -o "StrictHostKeyChecking=no" student@10.0.0.14 "sudo tc qdisc replace dev ens4 root netem
↪  delay 25ms"
```

This feature has been used in order to automate the test execution by means of a “dotest” bash script running on the host, following the approach adopted by ION [ION], the DTN suite developed by NASA-JPL. A further step of automation was introduced by writing a Python script to produce the final graphs.

4.1.1 The testbed layout

The tests were conducted on several virtual machines, whose layout is visible in Figure 4.1.

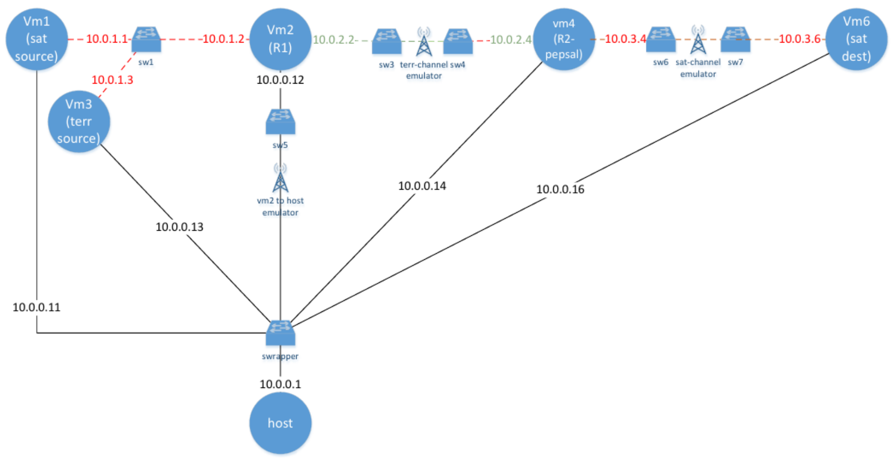

Figure 4.1: Layout used for the tests

The virtual machines VM1 and VM3 are normally used as senders, while the VM6 machine as a receiver. The layout also provides a channel emulator on the link between VM4 and VM6: generally this is designed to emulate a terrestrial link, but for the purposes of the tests shown in this chapter, this channel is considered ideal. The link between VM4 and VM6 aims to resemble different type of satellite communications (i.e. LEO and GEO satellites), with different propagation delay, bandwidths, and losses, as specified later test by test.

For the purposes of this thesis, the graphs are drawn using LibreOffice Calc and the data extracted by csv log files. Another possibility is the use of the online tool qvis [qvis] that uses QLOG files to plot both congestion related data (CWIN and RTT) and transmission related data (when a packet is sent, acknowledged and lost). It also offers a sequence of the exchange of packets between client and server.

4.1.2 The “dotest_MAC.sh” script

As stated before, the processes of launching the tests, configuration of the channels, and collection of results, was completely automated by a bash script launched on the host machine. This script, named `dotest_MAC.sh` after the names of people that contributed to it (Moffa, Andreetti, Cavallotti), consists of a series of loops that iterate on different parameters (e.g. delay, congestion control, packet error rate, etc.). At each iteration, the script launches clients and servers and modifies the channels, by means of the `tc netem` program (explained below). After all loops are done, the script copies the logs (in .csv format) from the virtual machines to a specific directory on the Host. Then, it injects the contents of the log files in .csv format, produced during the tests, into a .ods LibreOffice file, using a Python script and specific .ods file used as templates. These templates automatically generate the graphs when the .csv files are injected into them. This script simply iterates on a number of .csv files and then copies the contents into the sheets of the template file (that has been previously copied).

The setup of the satellite channel’s characteristics is accomplished by using the network tool *tc netem*. This simple but very strong tool is crucial for changing parameters like bandwidth, loss rate and delay. In order to change the characteristic of the channel, commands like the following are used:

```
sudo tc qdisc replace dev ens5 root netem delay 250ms rate 10Mbit limit 417 loss 0.01%
```

It is necessary to limit the buffer size by setting the limit option to the product of bandwidth and delay (divided by 1500, since it requires a number of packets) of the satellite channel. This is because otherwise the packets would be dropped too often, causing problems in some cases, especially when the delay and the bandwidth are big. It works by dropping incoming packets when the queue is full (*drop tail policy*).

4.1.3 Test performed

Three are the macro-tests that were performed in this thesis to evaluate QUIC performance in satellite communications:

- **First**: using one picoquicdemo connection directly between VM1 (sender) and VM6 (receiver), to evaluate performance of QUIC, both in GEO and LEO environments, in the absence of any competing traffic.
- **Second**: using two picoquic connections that compete with each other, using Hybla congestion control versus the other congestion control algorithms available in picoquic (NewReno, BBR, Cubic). The two senders are VM1, always using Hybla, and VM3, changing at every iteration; the receiver is VM6.
- **Third**: using one picoquicdemo connection, using Hybla congestion control, against one TCP connection that uses Cubic as congestion control. VM1 is the picoquicdemo sender, VM3 is the TCP sender, VM6 is the receiver.

The parameters over which the test iterates are:

- **Round Trip Time**: 30ms, to emulate a LEO satellite, and 500ms, to emulate a GEO satellite.
- **Data Rate**: 10, 25, 50 and $100\,\mathrm{Mb/s}$100 Mb/s.
- **Loss Rate**: 0.01 %, 0.1 %, 1 %, and 10 % only for the LEO satellite.
- **Congestion control**: NewReno, Cubic, BBR, Hybla.

The duration of the tests is 5 minutes, and they were performed using the time mode of the picoquicperf ALPN.

The metrics examined are:

- The variation of the value of the congestion window during the connection.
- The goodput of a connection, both using an average on 1-second window and the average since the beginning.
- When some form of competition is involved, measuring the friendliness is the aim of the experiments.

4.2 One QUIC satellite connection: impact of congestion control policy in the absence of competing traffic

The first macro-test examined in this section is the one without any form of competition. This means that on the virtual machine VM1 a picoquicdemo client is launched at every iteration of the loops.

4.2.1 LEO satellites

First, let us discuss the case in which the two picoquicdemo peers are inside a network that contains a LEO satellite. For all tests shown in this section, the delay is set to 30ms, since that is an average value of round-trip time between a communication from Earth to a LEO satellite. Figure 4.2, Figure 4.3, Figure 4.4 and Figure 4.5 show the evolution of the congestion window in the first 10 seconds in one of LEO satellite experiments. For this particular case, the delay is 30 ms, the bandwidth is set to 10 Mb/s, and the loss is 0.01 %.

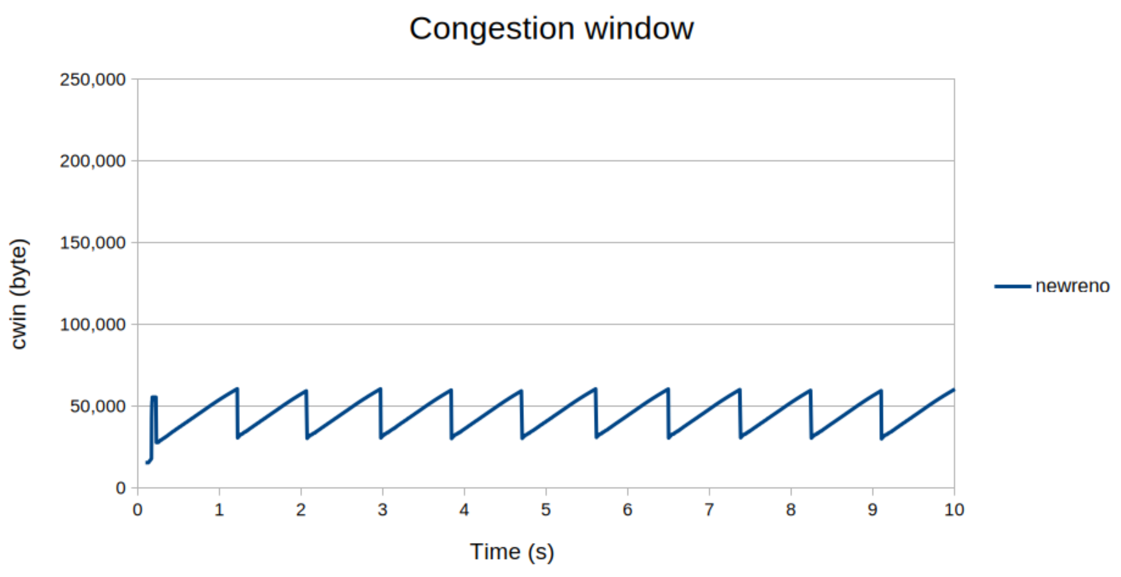

Figure 4.2: First 10 s of a connection that lasted 300 s, NewReno Congestion window, RTT: 30 ms, bandwidth: 10 Mb/s, loss: 0.01 %

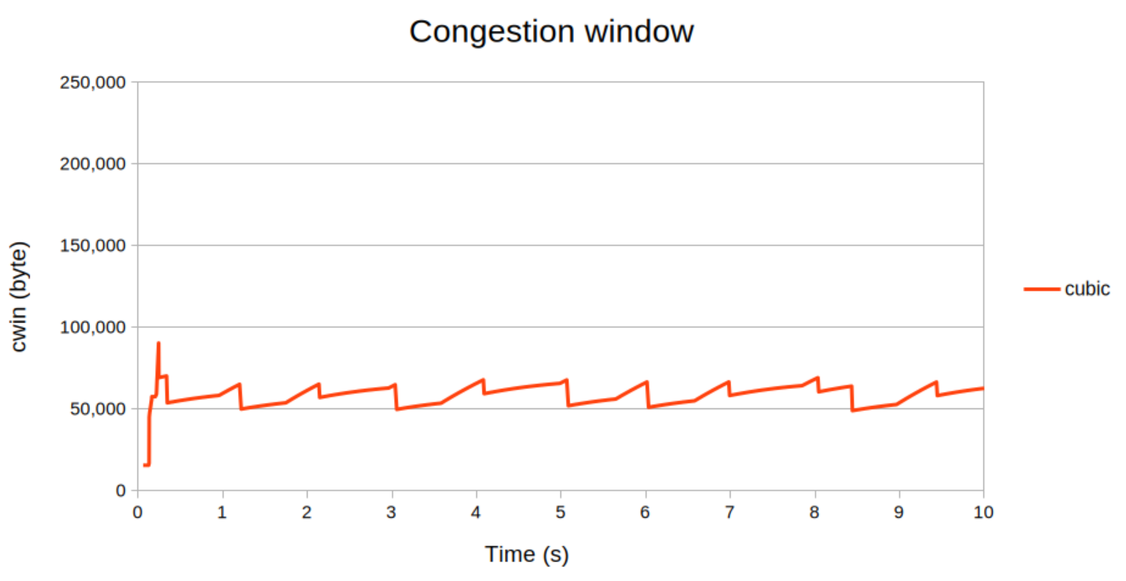

Figure 4.3: First 10 s of a connection that lasted 300 s, Cubic Congestion window, RTT: 30 ms, bandwidth: 10 Mb/s, loss: 0.01 %

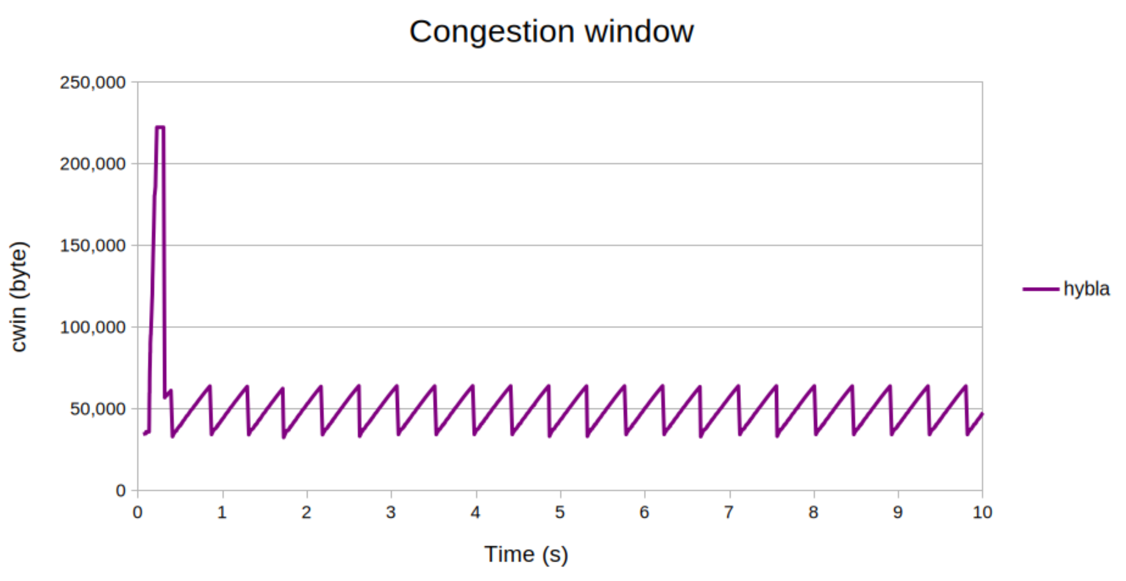

Figure 4.4: First 10 s of a connection that lasted 300 s, Hybla Congestion window, RTT: 30 ms, bandwidth: 10 Mb/s, loss: 0.01 %

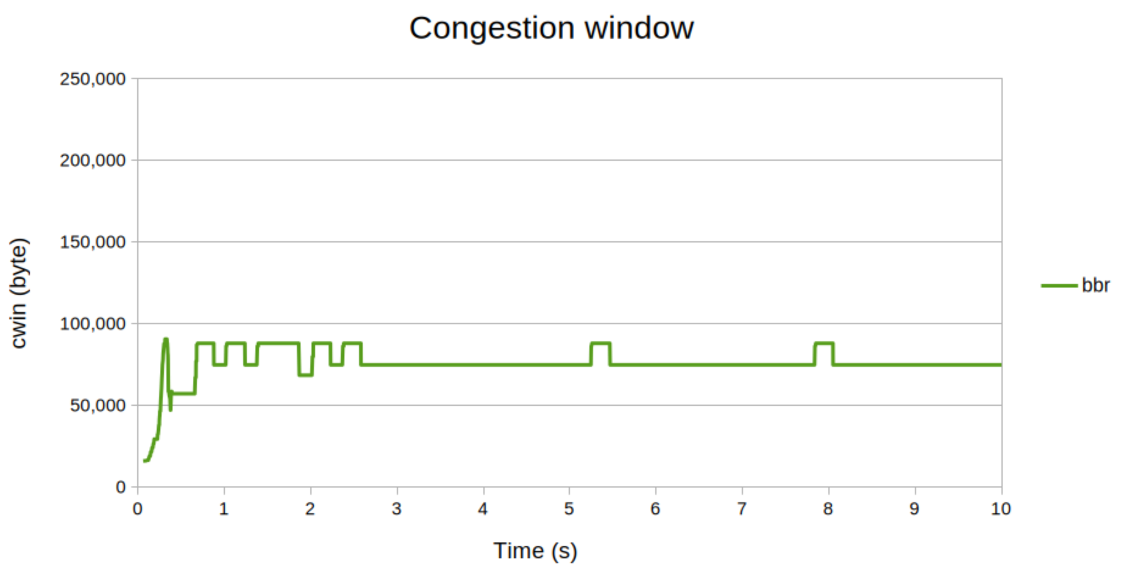

Figure 4.5: First 10 s of a connection that lasted 300 s, BBR Congestion window, RTT: 30 ms, bandwidth: 10 Mb/s, loss: 0.01 %

These graphs show the plot in an almost perfect channel. For example, NewReno and Hybla show the typical sawtooth shape. In general, all the algorithms have a plot that is very stable around almost the same value. This can be compared to the graphs in Figure 4.6, Figure 4.7, Figure 4.8 and Figure 4.9 where the only difference is that the loss rate is set to 10%.

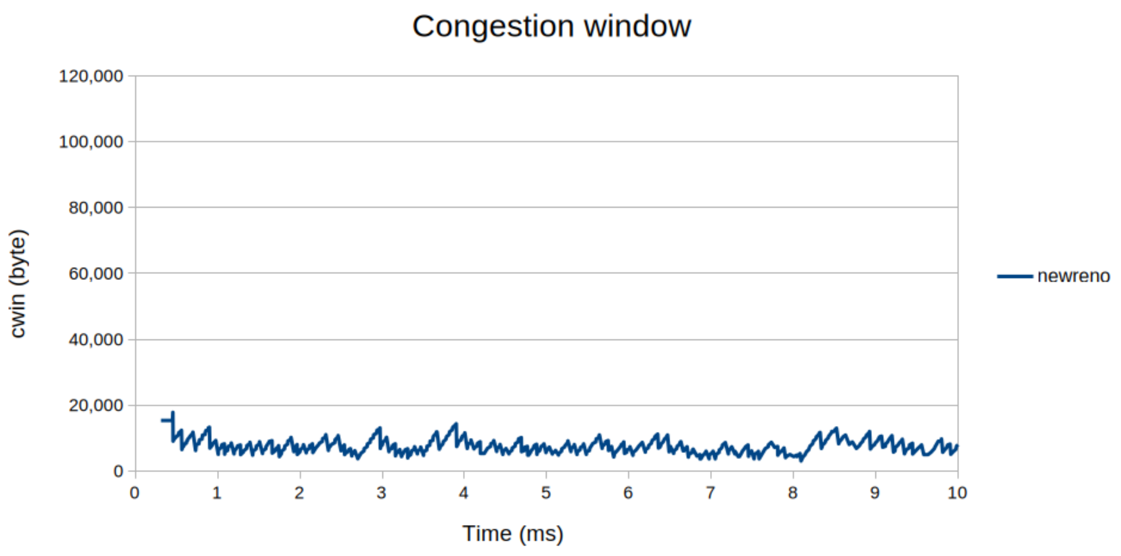

Figure 4.6: First 10 s of a connection that lasted 300 s, NewReno Congestion window, RTT: 30 ms, bandwidth: 10 Mb/s, loss 10 %

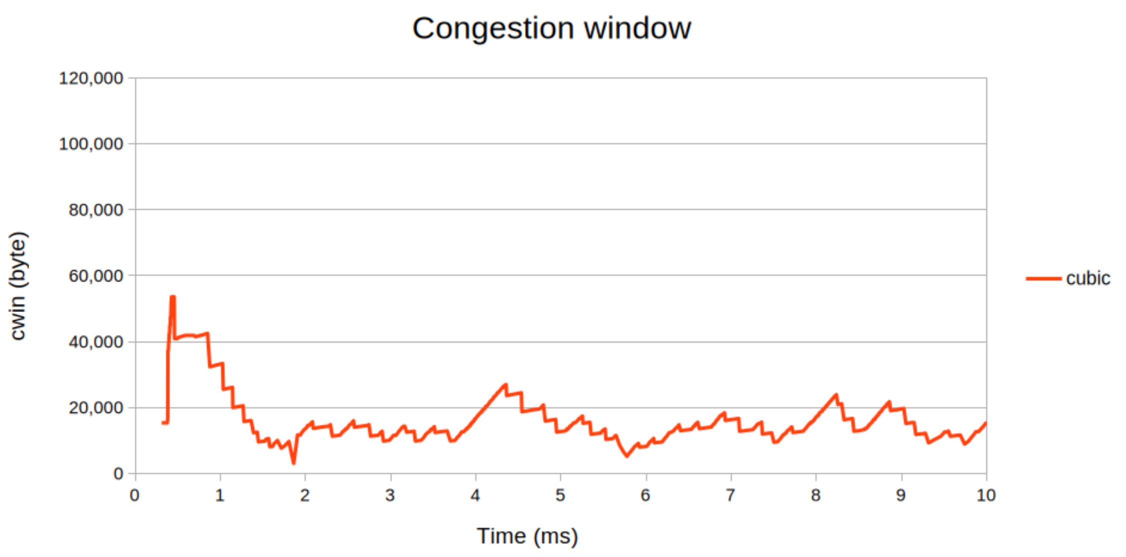

Figure 4.7: First 10 s of a connection that lasted 300 s, Cubic Congestion window, RTT: 30 ms, bandwidth: 10 Mb/s, loss 10 %

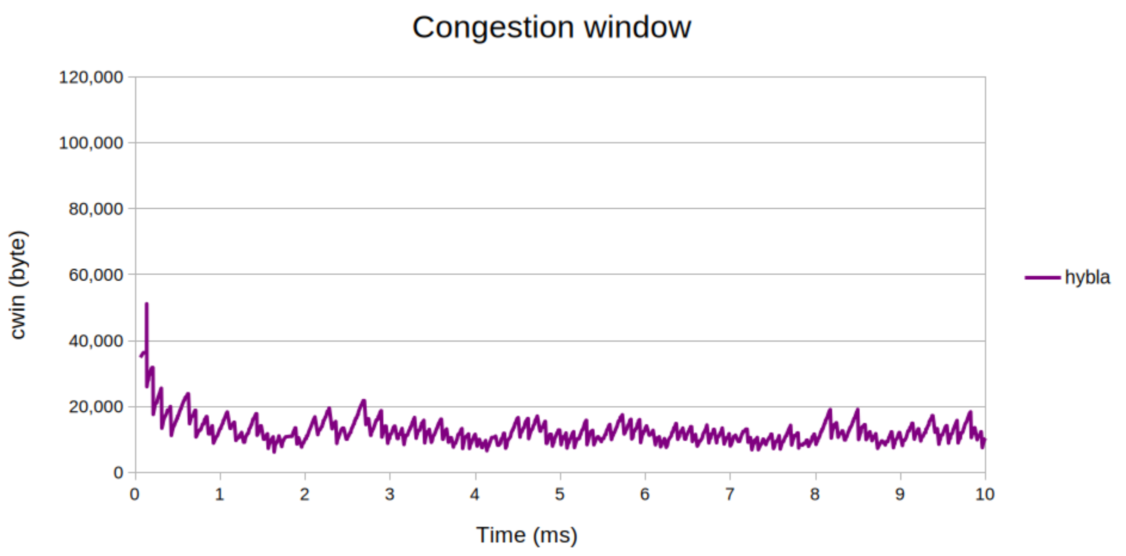

Figure 4.8: First 10 s of a connection that lasted 300 s, Hybla Congestion window, RTT: 30 ms, bandwidth: 10 Mb/s, loss 10 %

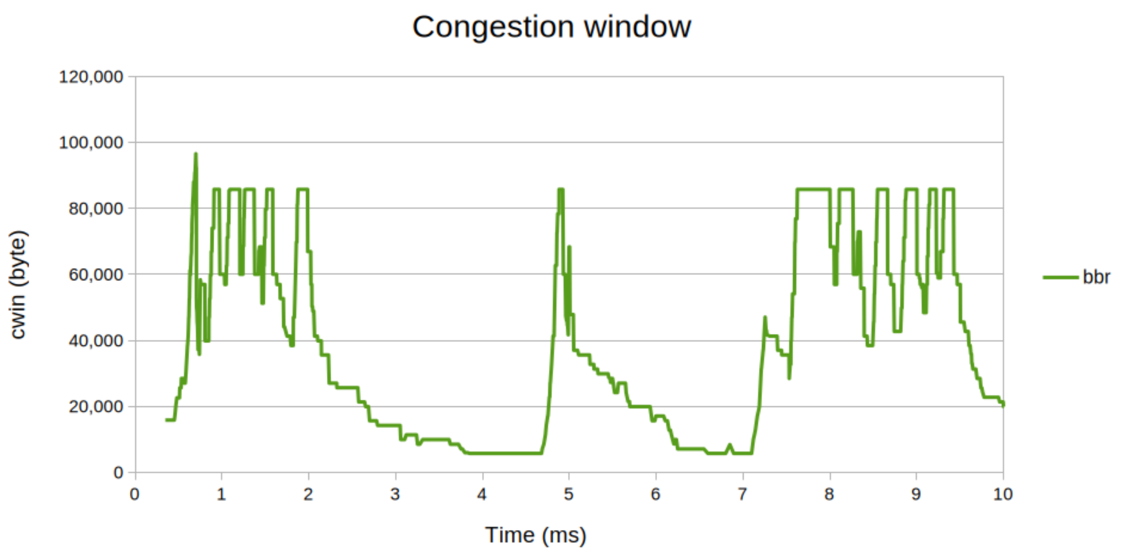

Figure 4.9: First 10 s of a connection that lasted 300 s, BBR Congestion window, RTT: 30 ms, bandwidth: 10 Mb/s, loss 10 %

In these graphs it's clear that the disruption brought by the loss rate is enough to change the aspect of the curves entirely. All the algorithms performed slightly worse than the previous experiment. For instance, NewReno, Cubic and Hybla all were gravitating around 5 000 bytes, whilst now the same value is reached only in the first moments of the connection. BBR, on the other hand, has a very unique behavior. The shape of the graph is due to the fact that this algorithm does not take into consideration lost packet when computing the congestion window value. Rather, it uses a specific method of probing to try and understand the dimension of the bandwidth, and then uses that estimation in its calculations.

To better understand the effects on the performance of these congestion control algorithms, it is useful to have a look at the value of the goodput during the connection. Figure 4.10 and Figure 4.11 show the plot of the goodput averaged a one second window, meaning that this is not an instantaneous sample, but rather an average taken on a one second interval. This type of representation was chosen because, in some cases, it can highlight behaviors in a more detailed way, in respect to an average computed over the whole duration. NewReno and BBR are not shown, but the former has a graph that is very similar to Hybla, and the latter has a graph that is very similar to Cubic.

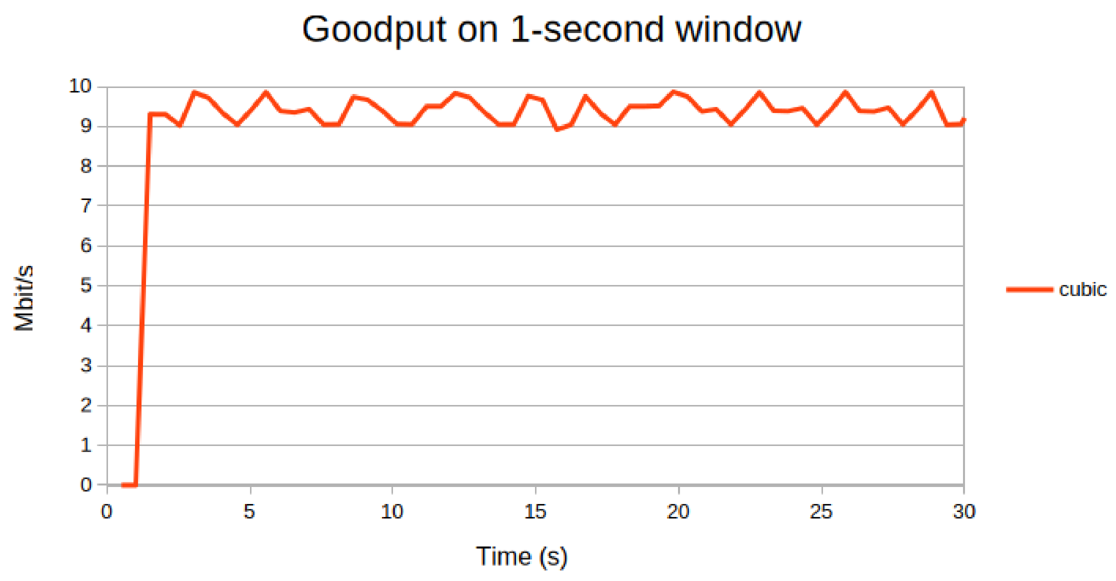

Figure 4.10: First 30 s of a connection that lasted 300 s, goodput Cubic, RTT: 30 ms, bandwidth: 10 Mb/s, loss 0.01 %

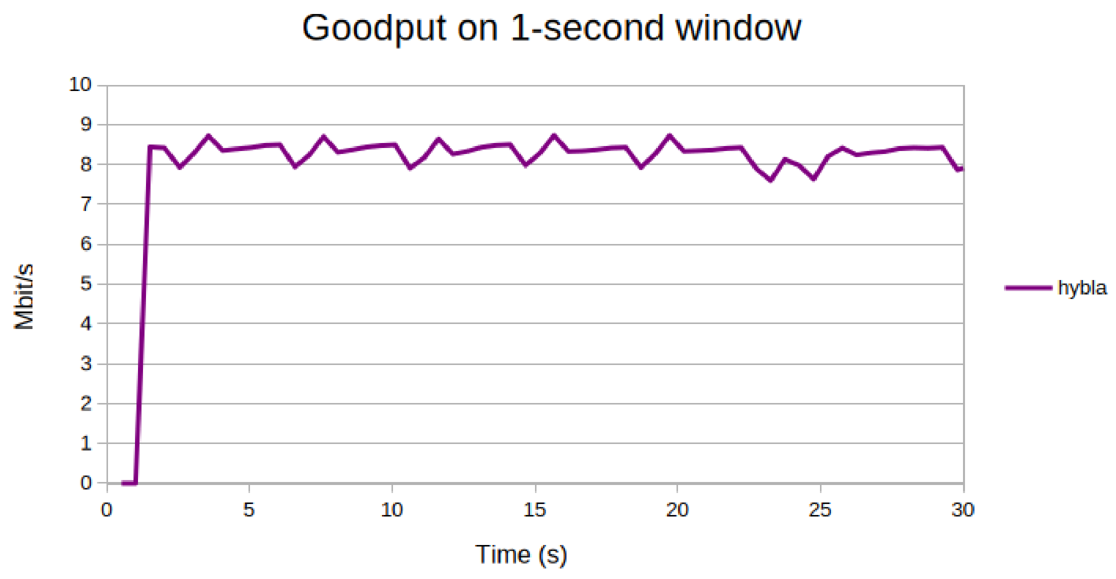

Figure 4.11: First 30 s of a connection that lasted 300 s, goodput Hybla, RTT: 30 ms, bandwidth: 10 Mb/s, loss 0.01 %

These graphs show that picoquic, in an environment that uses a LEO satellite and with very low loss rate, does not have any type of problems reaching a value that is very close to the maximum bandwidth of the channel. The same cannot be said about an environment with really high loss rate (10 %). As the congestion window graphs suggested, regardless of the congestion control used, picoquic's performance suffers a lot. For instance, Figure 4.12 and Figure 4.13 make it very obvious that the presence of those losses causes the connection's goodput to drop to extremely low levels.

To better visualize the importance of the loss rate in this type of connection, Figure 4.14 shows the different goodput, averaged on the whole duration of the connection, when changing the percentage of packet loss.

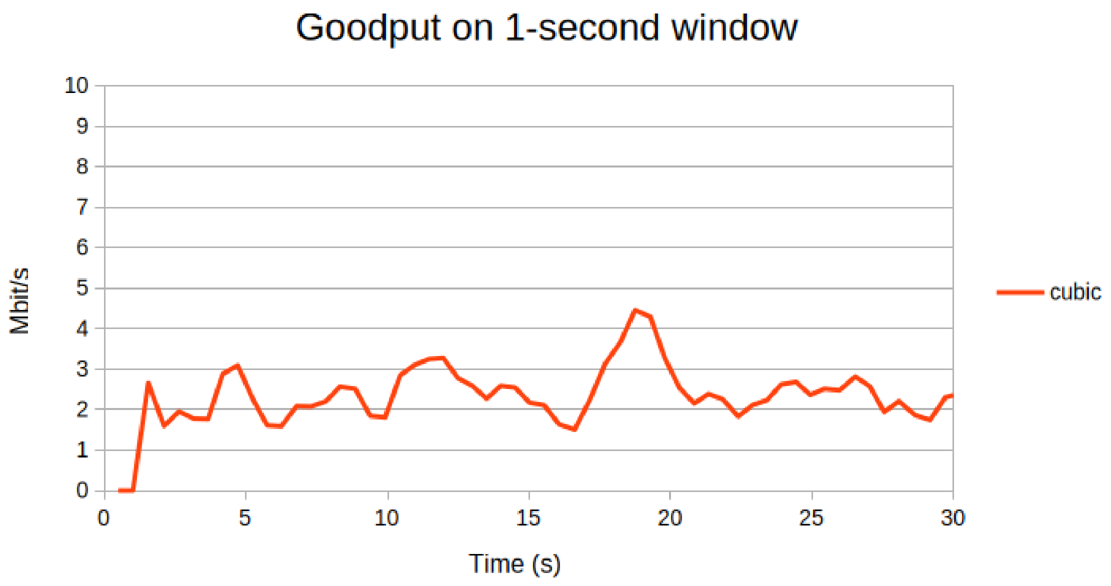

Figure 4.12: First 30 s of a connection that lasted 300 s, goodput Cubic, RTT: 30 ms, bandwidth: 10 Mb/s, loss 10 %

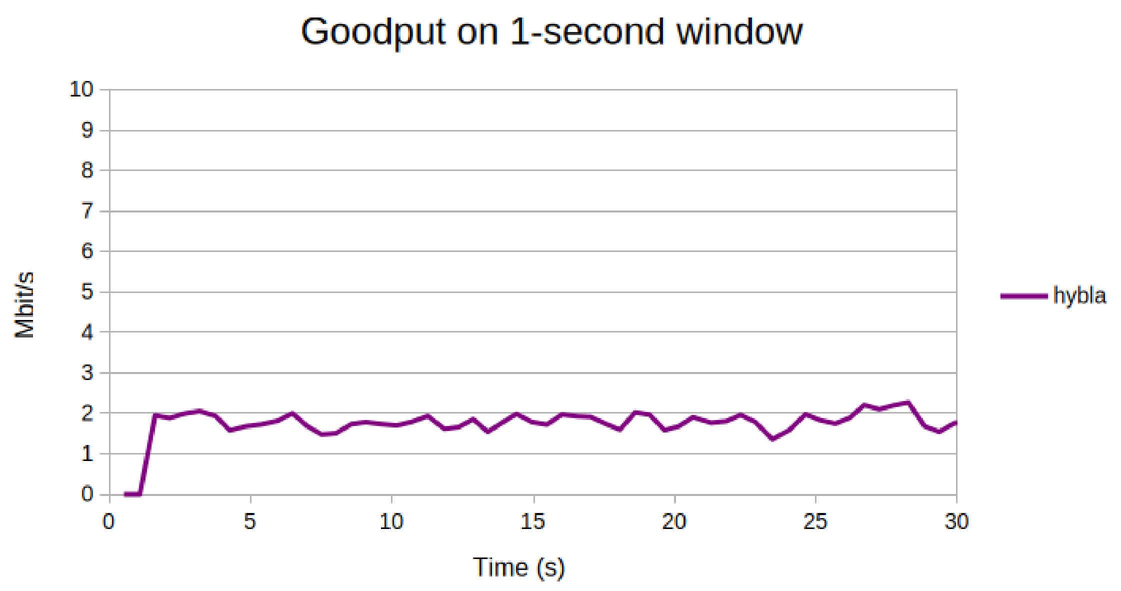

Figure 4.13: First 30 s of a connection that lasted 300 s, goodput Hybla, RTT: 30 ms, bandwidth: 10 Mb/s, loss 10 %

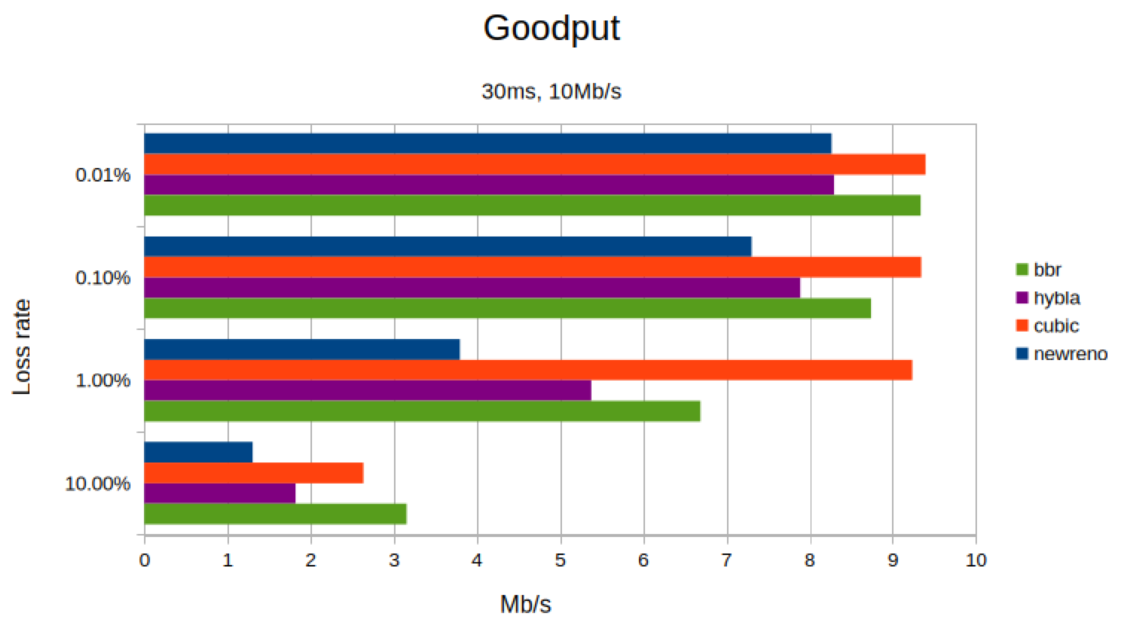

Figure 4.14: Goodput for different variations of loss rate, RTT: 30 ms, bandwidth: 10 Mb/s

4.2.2 GEO satellites

Now let's analyze the case of a GEO satellite network. For all tests shown in this section, the delay is set to 500 ms, since that is an average value of round-trip time between a communication from Earth to a GEO satellite. The Figure 4.15, Figure 4.16, Figure 4.17 and Figure 4.18 show what happens in a GEO environment while having a bandwidth of 10 Mb/s and a loss rate of 0.01 %.

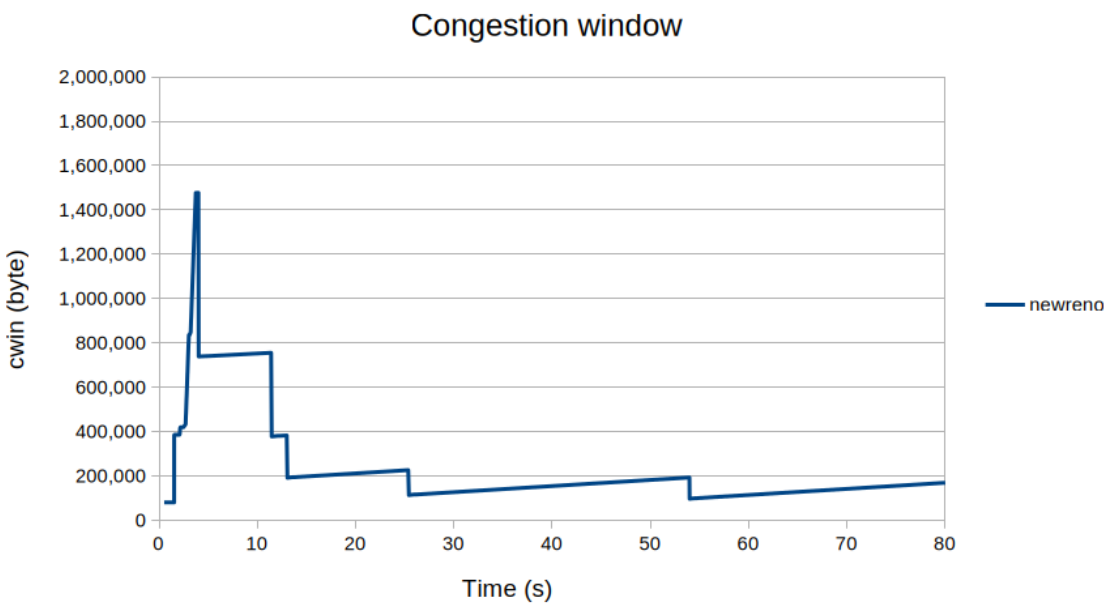

Figure 4.15: First 80 s of a connection that lasted 300 s, congestion window NewReno, RTT: 500 ms, bandwidth: 10 Mb/s, loss 0.01 %

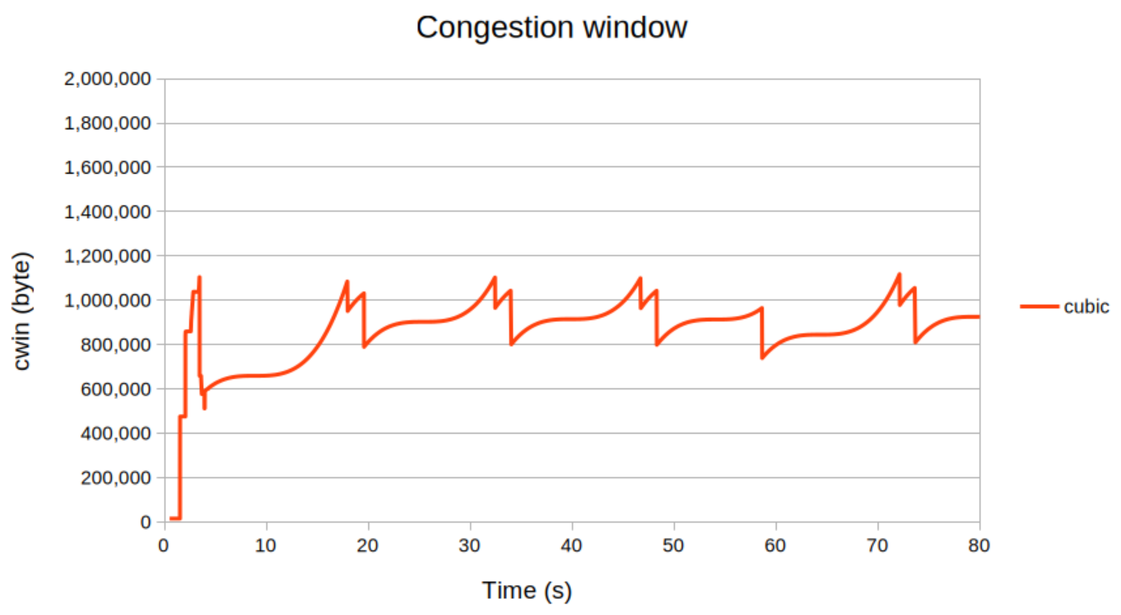

Figure 4.16: First 80 s of a connection that lasted 300 s, congestion window Cubic, RTT: 500 ms, bandwidth: 10 Mb/s, loss 0.01 %

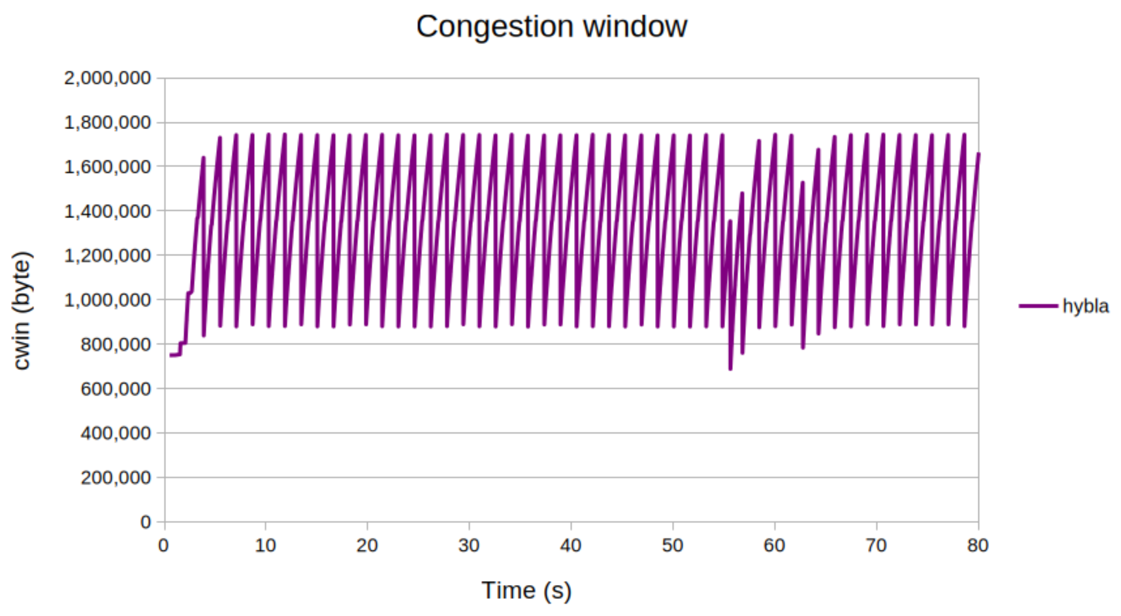

Figure 4.17: First 80 s of a connection that lasted 300 s, congestion window Hybla, RTT: 500 ms, bandwidth: 10 Mb/s, loss 0.01 %

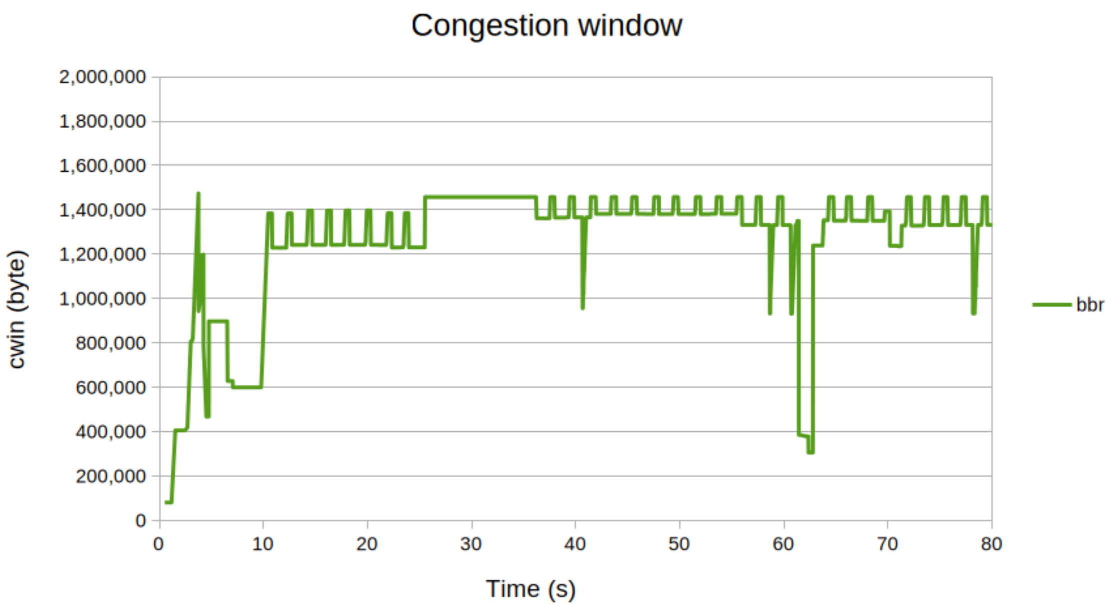

Figure 4.18: First 80 s of a connection that lasted 300 s, congestion window BBR, RTT: 500 ms, bandwidth: 10 Mb/s, loss 0.01 %

Comparing these to the corresponding graph obtained in the LEO connection, some curious aspects can be deducted. First, it's clear that NewReno is the one that suffers the most. This is because when the algorithm enters the congestion avoidance phase, it increases the window every time an ACK is received. Since the round trip time is so big, it takes more round trips for the algorithm to reach the same value. Meanwhile, Hybla is able to reach higher values more quickly, thanks to the acceleration provided by the rho factor. This is exactly the expected behavior of this congestion control. Lastly, Cubic and BBR seems to have a behavior that is inline with the specification of their algorithms.

This results can be then compared to the graphs containing the plot of the goodput. Figure 4.19 shows that cubic has an harder time keeping the goodput at a stable value. Differently, Figure 4.20 makes it clear that Hybla is able to stabilize itself after a short period of time. This shows that having Hybla as a congestion control algorithm can be the best choice when dealing with long delays. Further proof of this can be seen by comparing Hybla's graph with NewReno's [Figure 4.21]: the latter suffer incredibly with longer delays. BBR's graph is not shown but its plot is very similar to Cubic's.

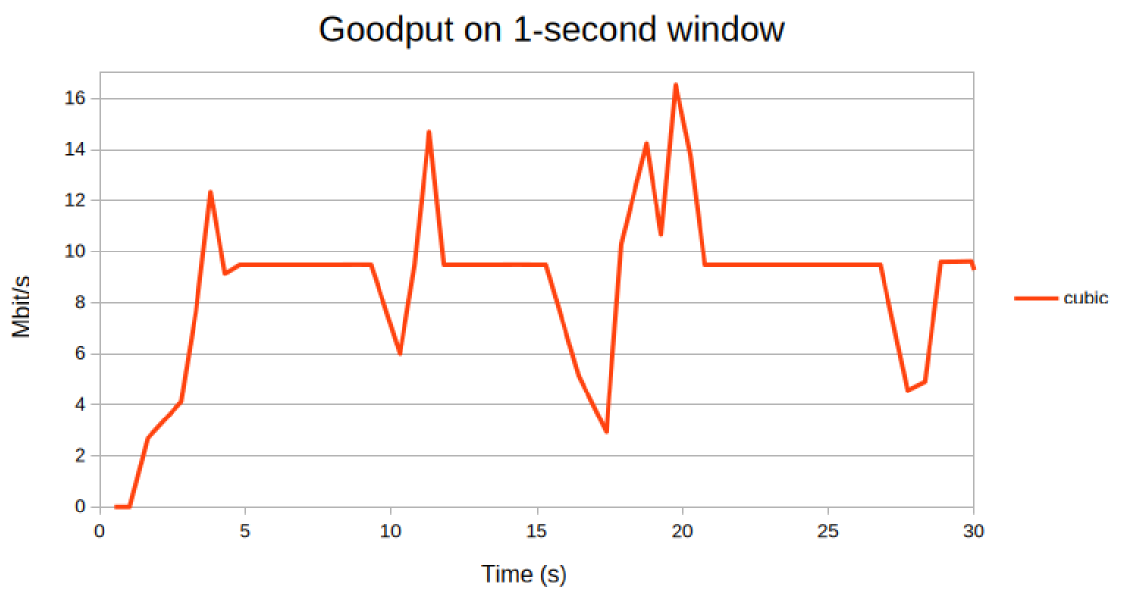

Figure 4.19: First 30 s of a connection that lasted 300 s, goodput Cubic, RTT: 500 ms, bandwidth: 10 Mb/s, loss 0.01 %

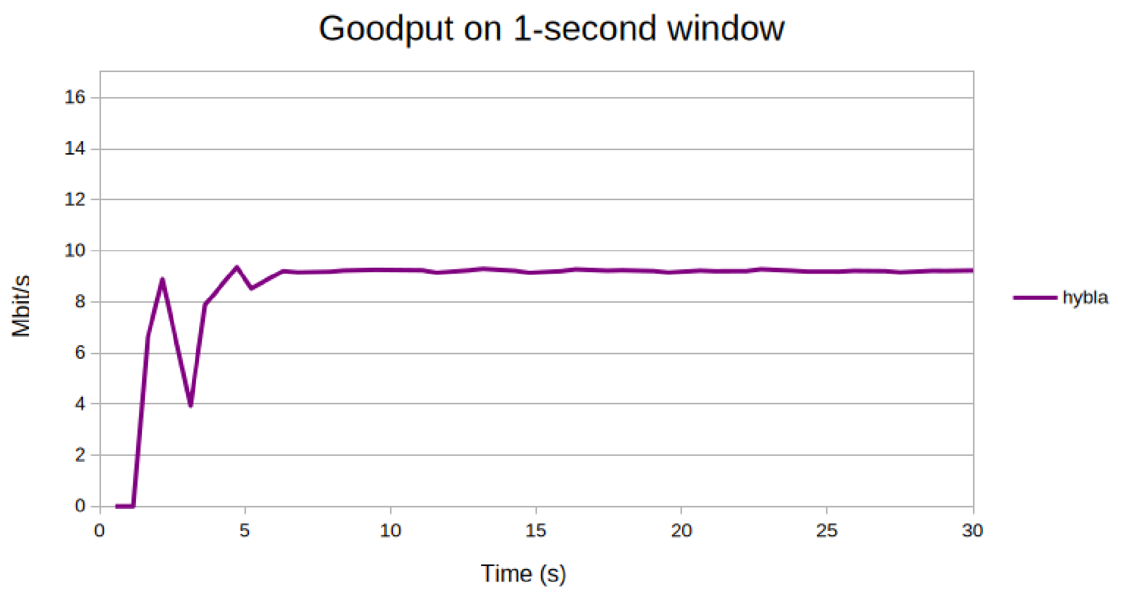

Figure 4.20: First 30 s of a connection that lasted 300 s, goodput Hybla, RTT: 500 ms, bandwidth: 10 Mb/s, loss 0.01 %

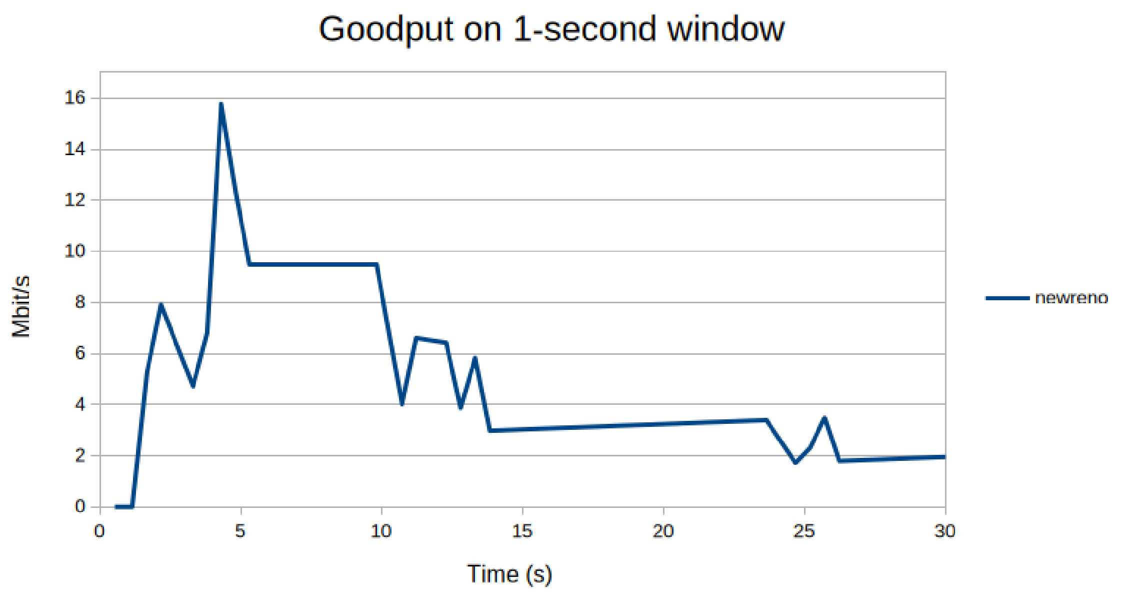

Figure 4.21: First 30 s of a connection that lasted 300 s, goodput NewReno, RTT: 500 ms, bandwidth: 10 Mb/s, loss 0.01 %

The “jumps” that are visible in the goodput graphs, both in the LEO and GEO cases, are due to a sudden increase in the number of bytes that are delivered to the application (on receiver side). This can be because, for example, a packet was missing and the retransmission of it blocked the delivery of a lot of data that was queued on receiver’s side. As soon as the missing packet is delivered, all the data that was held in a waiting queue is immediately delivered to the application. In Hybla, such problem is a lot less visible because the algorithm implements a system of pacing between packets that block the sender’s application to over-shoot them.

Finally, let us discuss about the overall trend of the goodput. Aside for NewReno, all congestion control algorithms led to the reach of the maximum bandwidth available, in the experiment that had 10Mb/s as a maximum [Figure 4.22]. Even varying the loss rate, picoquicdemo very easily reached the best value.

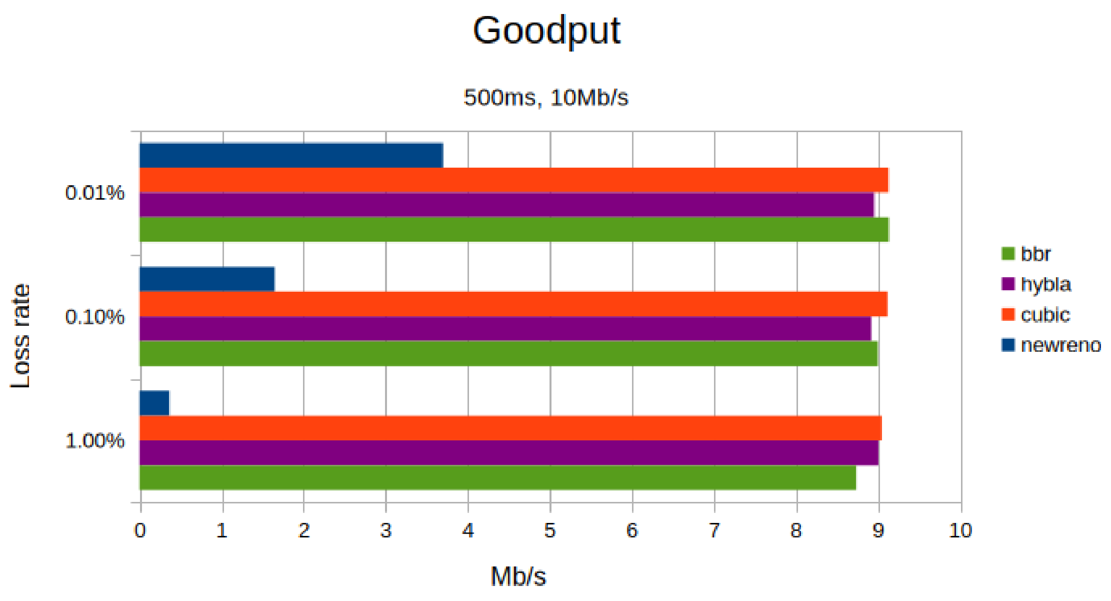

Figure 4.22: Goodput for different variations of loss rate, RTT: 500 ms, bandwidth: 10 Mb/s

4.2.2.1 Bandwidth of 100 Mb/s

A more interesting case is the one with bandwidth set to 100 Mb/s, shown in Figure 4.23. In this graph it is possible to see that, varying the loss rate, almost all of the algorithms suffer a worsening of their performance. The only one that is capable of reaching a value that is very close to the maximum bandwidth is BBR, and its averaged value does not change (because, as stated before, this algorithm does not take into consideration packet loss).

These results prove that Hybla as a congestion control can still provide sufficient performance in a GEO satellite environment compared to the other algorithms, when the losses are relatively low.

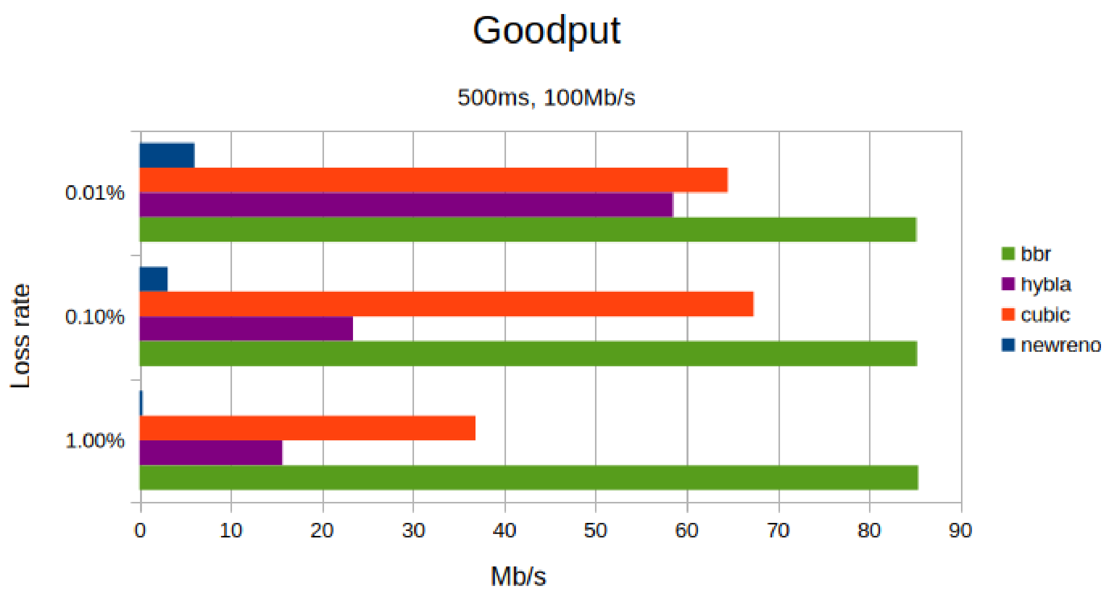

Figure 4.23: Goodput for different variations of loss rate, RTT: 500 ms, bandwidth: 100 Mb/s

4.3 Two QUIC satellite connections: congestion control friendliness

This section focuses on the results of the tests that involved some kind of competition. In particular, the aim is to analyze friendliness of the tools used. The latter was measured by launching two picoquicdemo instances on VM1 and VM3: the machine 1 only used Hybla, while the machine 3 used the other congestion control algorithms set inside the loop of the bash script. The friendliness refers to the behavior of picoquic when other type of traffic is present on the network; for instance, this was measured by launching competitive TCP traffic from the VM3, while on the VM1 a picoquicdemo client that uses Hybla was launched at every iteration.

4.3.1 LEO satellites

Let us start from considering the case in which the machines operate using a LEO satellites in the network.

The graph in Figure 4.24 shows the plot of the congestion window when Hybla is competing with NewReno in a LEO environment, using a bandwidth of 10 Mb/s and a loss of 0.01 %. It is clear that the plot follows the sawtooth shape typical to these algorithms. To better understand if this competition is fair, the graph in Figure 4.25 shows the goodput of both machines using an average value on a 1-second window. One can easily see that the value reached by Hybla, even though it is higher, is still close to the middle point. It is also noticeable that when one line increases, the other one decreases; this could be a strong indication of friendliness.

To better analyze the friendliness, one can use the Jain fairness index. This index is used to understand how a specific resource is shared among different users. In this case the shared resource is the bandwidth. The formula is the following:

$$J = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \cdot \sum_{i=1}^{n} x_i^2}$$

In this case, the average final goodput on both machines is 2.9755 Mb/s for NewReno, and 5, 7253 Mb/s for Hybla. Putting those numbers inside the formula gives an index of 0.9091, and since the closer a number is to 1, the fairer the two components shared the resource, we can conclude that the bandwidth was shared equally.

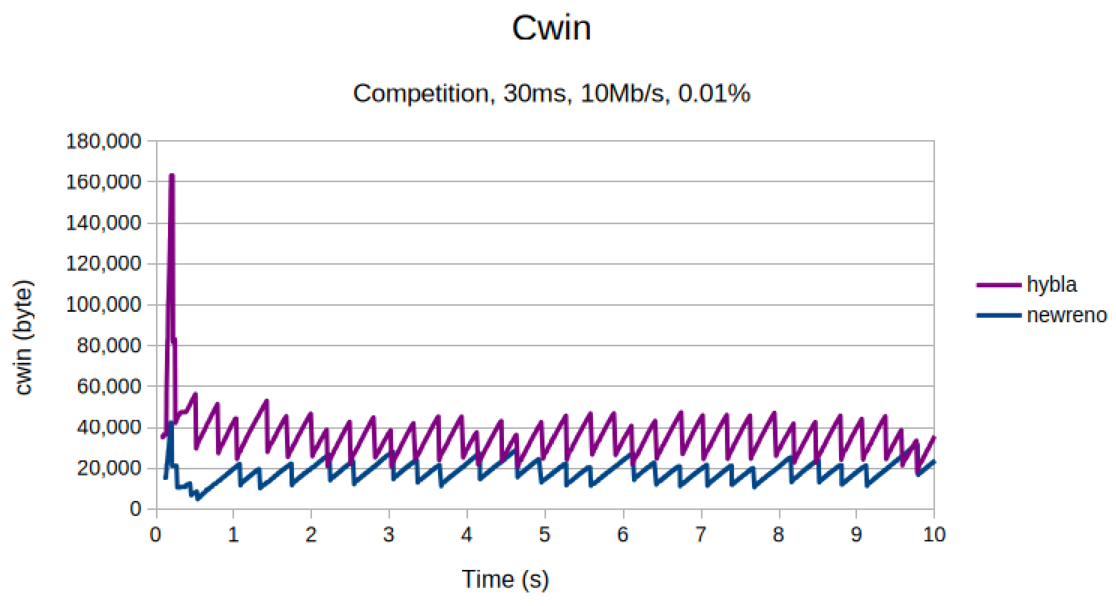

Figure 4.24: First 10 s of a connection that lasted 300 s, Competition between Hybla and NewReno, RTT: 30 ms, bandwidth 10 Mb/s, 0.01 % loss

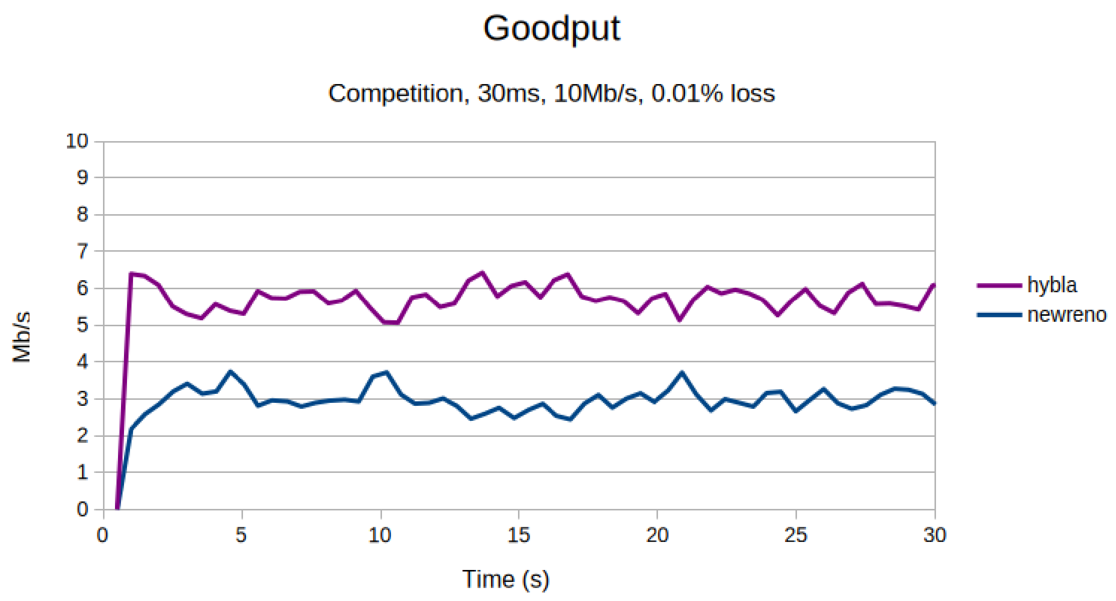

Figure 4.25: First 30 s of a connection that lasted 300 s, Competition between Hybla and NewReno, RTT: 30 ms, bandwidth 10 Mb/s, 0.01 % loss

Another example in the LEO tests is the case where Hybla competed against Cubic. The Figure 4.26 shows the plot of the two congestion windows. At first sight, it is clear that the values are very similar throughout the connection. By looking at the goodput graph in Figure 4.27 it's clear that they share a very similar value, and, also in this case, when one line increases the other one decreases.

Let's calculate the Jain fair index again. The final average goodput for Cubic is 5.1139 Mb/s and for Hybla is 4.2783 Mb/s. With these numbers the index is equal to 0.9921, which completely confirms that these two algorithms are fair with one another.

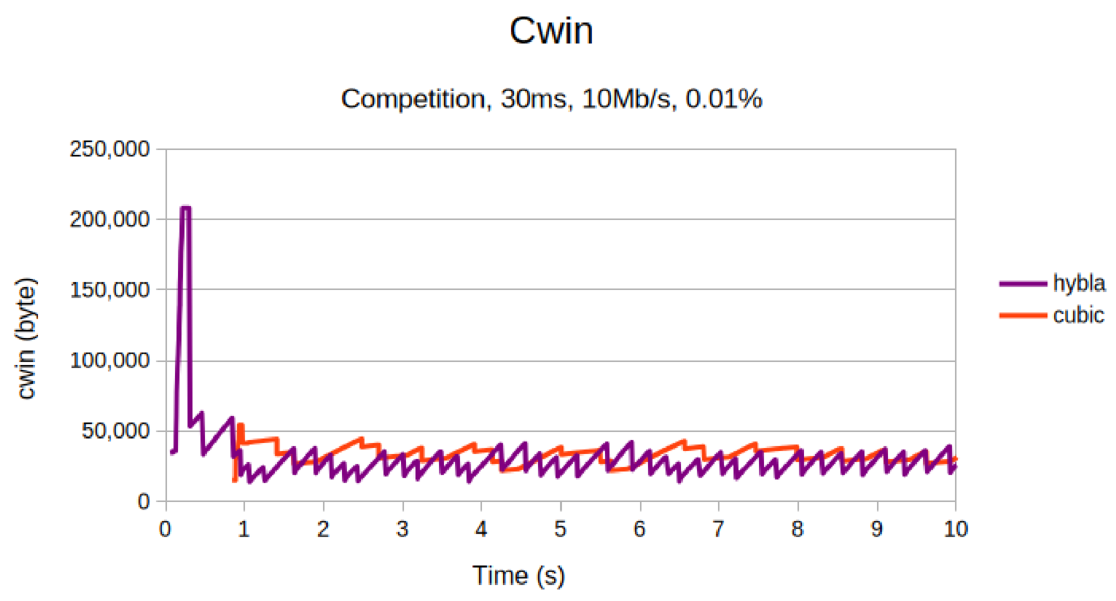

Figure 4.26: First 10 s of a connection that lasted 300 s, Competition between Hybla and Cubic, RTT: 30 ms, bandwidth 10 Mb/s, 0.01 % loss

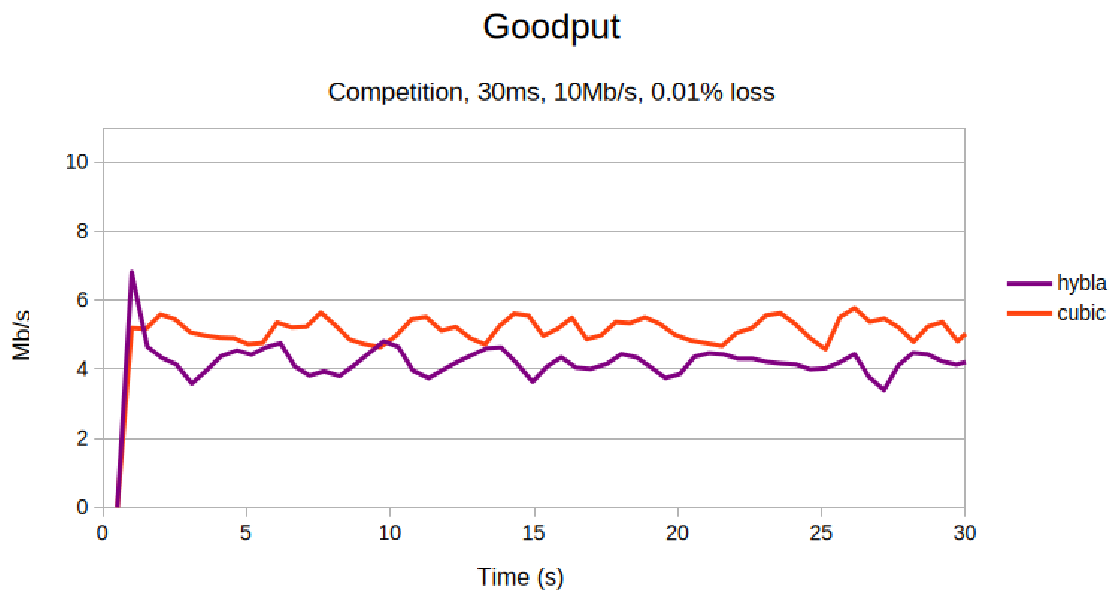

Figure 4.27: First 30 s of a connection that lasted 300 s, Competition between Hybla and Cubic, RTT: 30 ms, bandwidth 10 Mb/s, 0.01 % loss

As a last example, let's consider the case in which the competing traffic starts 5 seconds after the one from VM1. First the Figure 4.28 shows the plot of the congestion windows. As expected, once the second connection starts the congestion windows of the first suffer a downgrade. The same happens in the graph showing the goodput, in Figure 4.29. These results show that even though the second connection enters in action when the first one has almost filled the bandwidth, they immediately share it very fairly. This result suggests that QUIC is very friendly as a protocol.

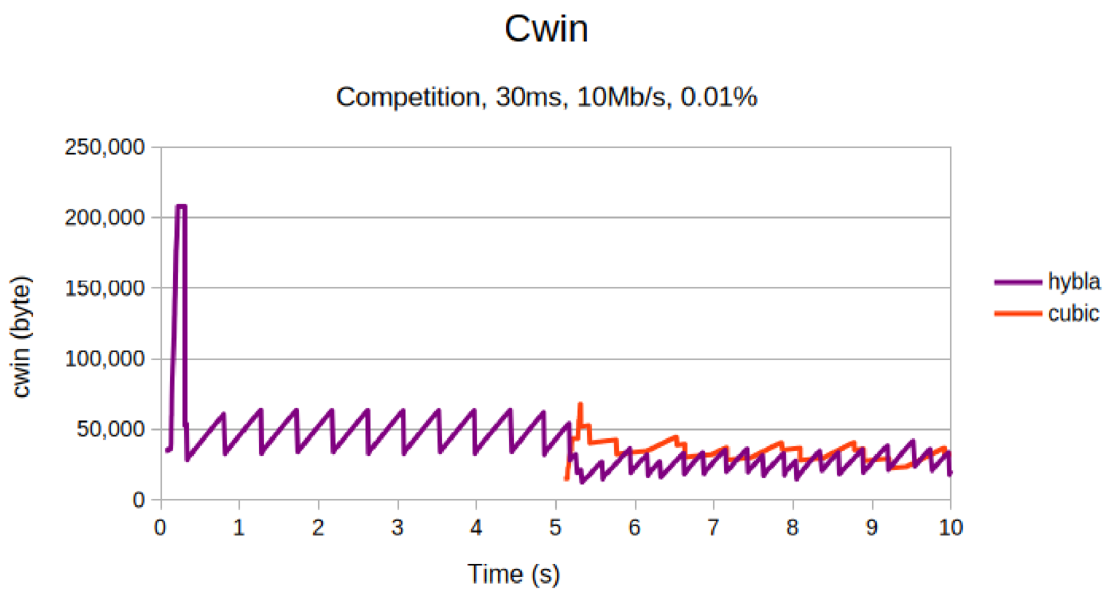

Figure 4.28: First 10 s of a connection that lasted 300 s, Competition between Hybla and Cubic (started 5 s after), RTT: 30 ms, bandwidth 10 Mb/s, 0.01 % loss

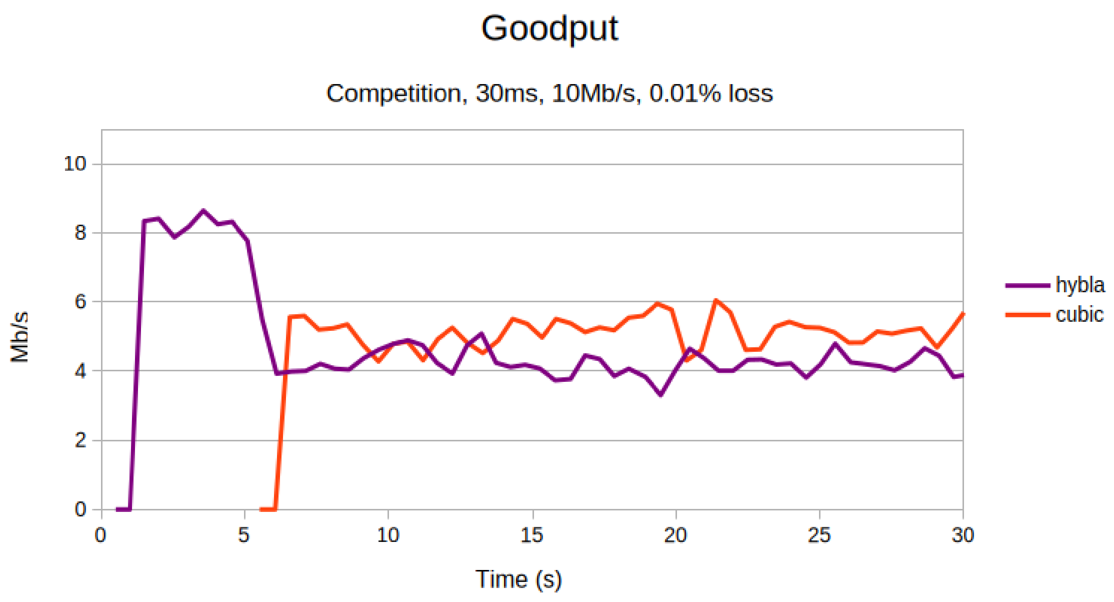

Figure 4.29: First 30 s of a connection that lasted 300 s, Competition between Hybla and Cubic (started 5 s after), RTT: 30 ms, bandwidth 10 Mb/s, 0.01 % loss

4.3.2 GEO satellites

For what concerns the GEO experiments, the results are very different. For instance, let's begin by looking at the graphs for the congestion window of Hybla versus NewReno shown in Figure 4.30. At first sight it is easy to imagine that NewReno suffers the most in these conditions. This is confirmed by the graph showing the goodput, in Figure 4.31, where it is clear that NewReno get completely crushed by both the high round trip time and also Hybla. The average goodput for NewReno is 0.1838 Mb/s and the average goodput for Hybla is 8.7872 Mb/s, so this gives an index of fairness of 0.5209, which is a lot lower than in the case of a LEO connection.

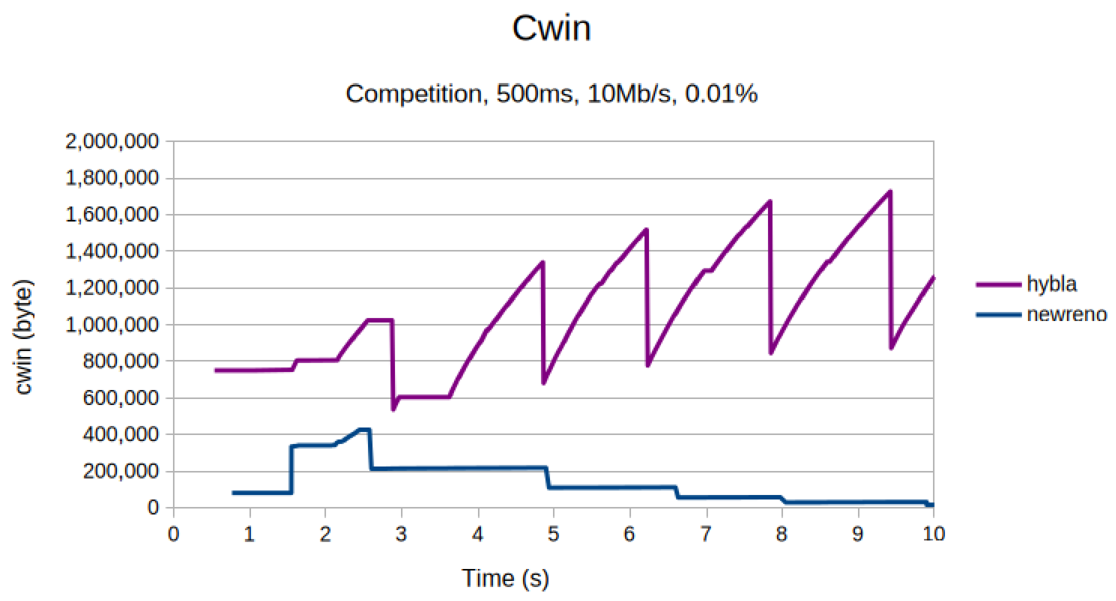

Figure 4.30: First 10 s of a connection that lasted 300 s, Competition between Hybla and NewReno, RTT: 500 ms, bandwidth 10 Mb/s, 0.01 % loss

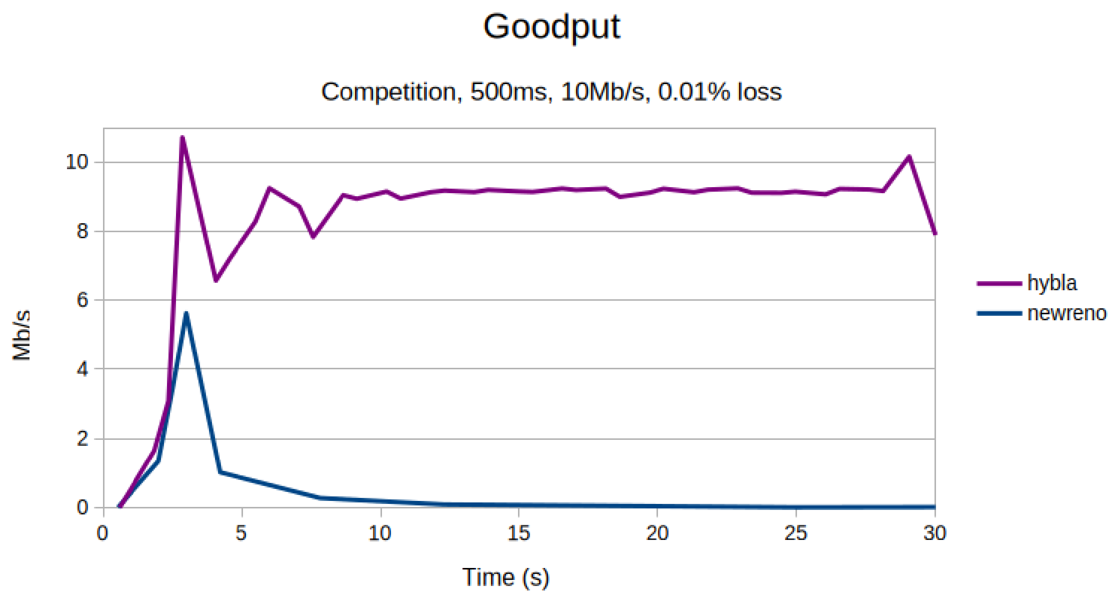

Figure 4.31: First 30 s of a connection that lasted 300 s, Competition between Hybla and NewReno, RTT: 30 ms, bandwidth 500 Mb/s, 0.01 % loss

Finally, let's analyze the case where the competition is between Hybla and Cubic, in a GEO environment. The Figure 4.32 shows the plot of the congestion window; Cubic seems to suffer a bit the presence of a long RTT, while the shape of Hybla's graph is very similar to the one with NewReno. The graph in Figure 4.33 shows the plot of the goodput in this connection. It is clear in this experiment that Cubic did suffer the presence of a long delay. The average goodput for Cubic is 0.9513 Mb/s, while Hybla's is 8.1155 Mb/s; this gives and index of 0.6156, which is lower than the one in a LEO connection, but higher than NewReno's in a GEO environment.

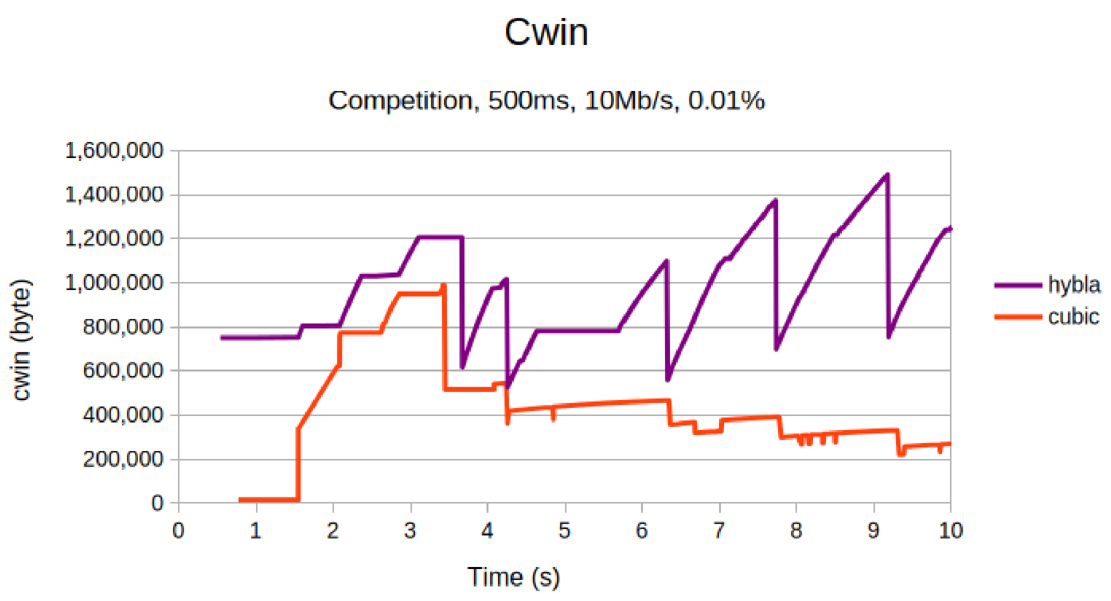

Figure 4.32: First 10 s of a connection that lasted 300 s, Competition between Hybla and Cubic, RTT: 500 ms, bandwidth 10 Mb/s, 0.01 % loss

As a last example, refer to Figure 4.34 to see the plot of the competition between QUIC Hybla vs QUIC Hybla in a GEO experiment. In this case, the metric to analyze is the fairness of the congestion control. From the initial 30 s it seems that the two plots are extremely similar, which makes sense since they are from the same congestion control. A more detailed result about the fairness can be obtained from the Jain fairness index. The first Hybla connection has am average goodput of 4.085 Mb/s, while the second is 4.1036 Mb/s. This gives an index of 0.9999, that signifies a perfect fairness.

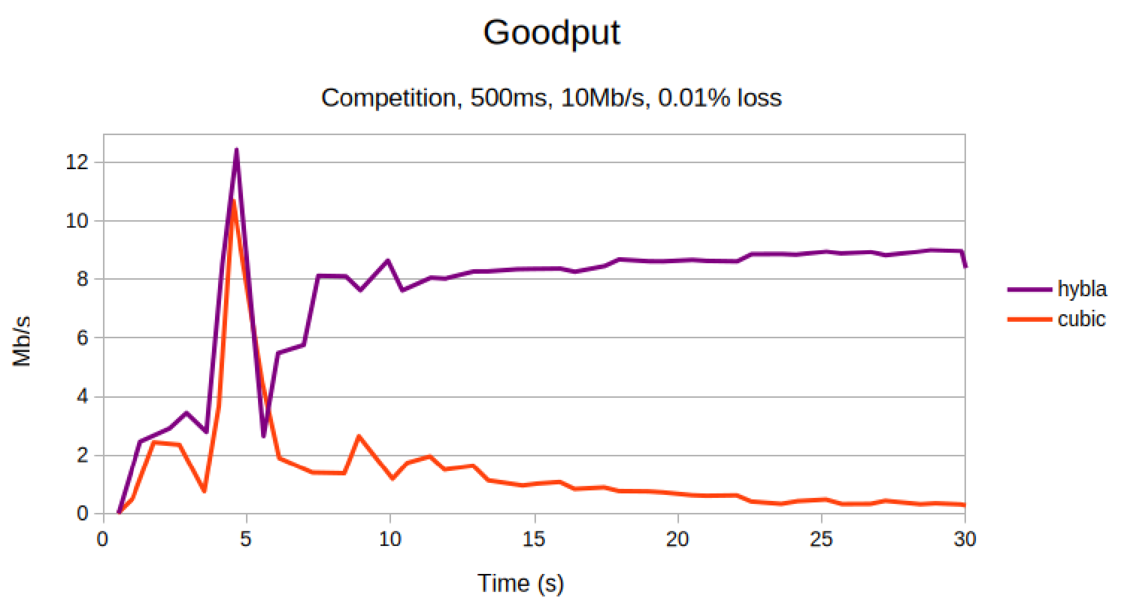

Figure 4.33: First 30 s of a connection that lasted 300 s, Competition between Hybla and Cubic, RTT: 30 ms, bandwidth 500 Mb/s, 0.01 % loss

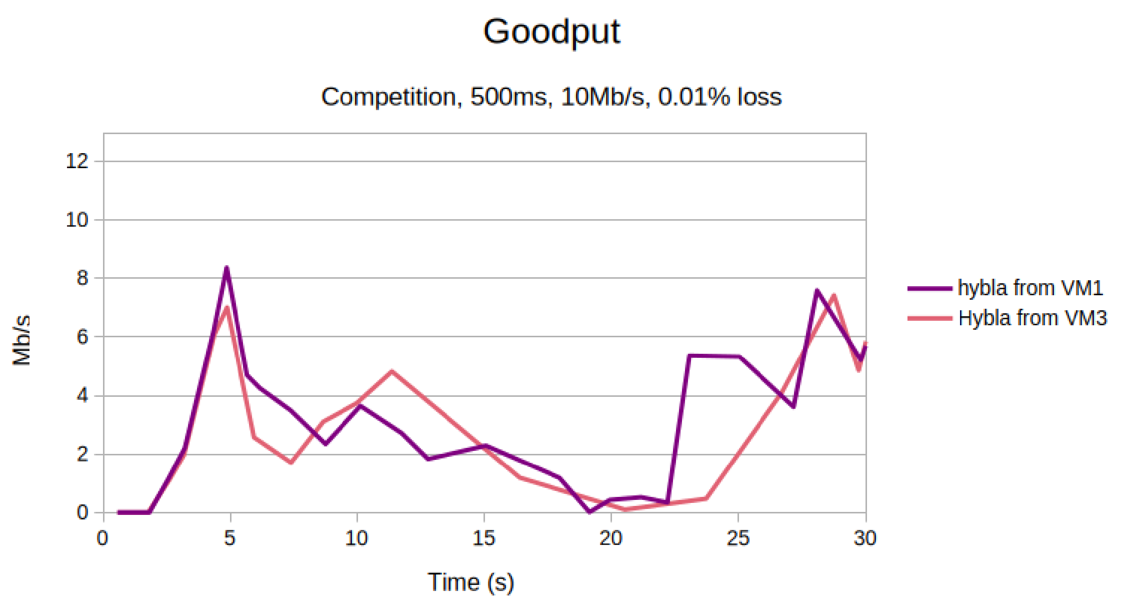

Figure 4.34: First 30 s of a connection that lasted 300 s, Competition between Hybla and Hybla, RTT: 30 ms, bandwidth 500 Mb/s, 0.01 % loss

4.4 One QUIC satellite connection and one TCP connection

To measure the friendliness of picoquic while using Hybla as congestion control, the tests were run by launching a picoquicdemo client on the VM1 and a picoquicdemo server on the VM6. Meanwhile, to generate TCP traffic, iperf was used [Iperf]:

- On the machine 6 a server is launched and stopped at every iteration, using two options to specify how often to produce the logs and what kind of file to produce (in this case, .csv):
  ```
  iperf -s -y C -i 0.5
  ```
- On the machine 3 a client is launched with the following command:
  ```
  iperf -c 10.0.3.6 -t 0
  ```
  With these parameters the client will communicate with a server located at the IPv4 address 10.0.3.6 (VM6), execute and then stop only when receiving a SIGKILL signal (launched when the picoquicdemo client is done transmitting).

It is important to know that the TCP traffic uses the default congestion control algorithm usually found in Linux distributions, which is Cubic.

4.4.1 LEO satellites

Figure 4.35 shows the plot of the congestion window reached by Hybla in the LEO experiments. The aspect of this plot is very similar to the one obtained from the competition between Hybla and Cubic using picoquicdemo, but in this case the trend of the graph seems a lot more stable. Unfortunately, the plot of the congestion window when using a TCP connection is not easy to obtain, since it would require to modify the kernel of the Linux machine in order to print or save somewhere the value of the congestion window.

The graph in Figure 4.36 shows how only at the beginning the TCP traffic is able to compete fairly with the picoquic client. Even during the execution, there are only a few places on the graph where there seems to be a mutual exchange of bandwidth (one increases, and the other decreases), and besides those it seems that the two clients are not sharing equally the bandwidth.

The average value of the goodput for Hybla is $6.6948\,\mathrm{Mb/s}$, while for TCP is $1.8847\,\mathrm{Mb/s}$ 1.8847, so the Jain index results equal to 0.7608, which is not incredibly high but neither too low.

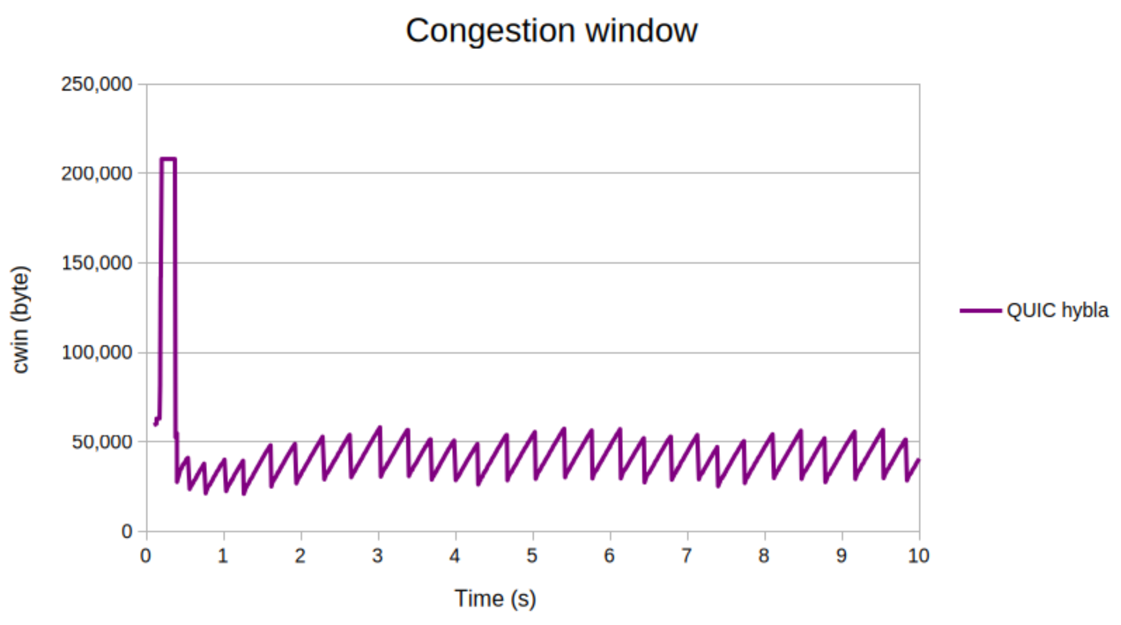

Figure 4.35: First 10 s of a connection that lasted 300 s, Competition between QUIC Hybla and TCP Cubic, RTT: 30 ms, bandwidth: 10 Mb/s, 0.01 % loss

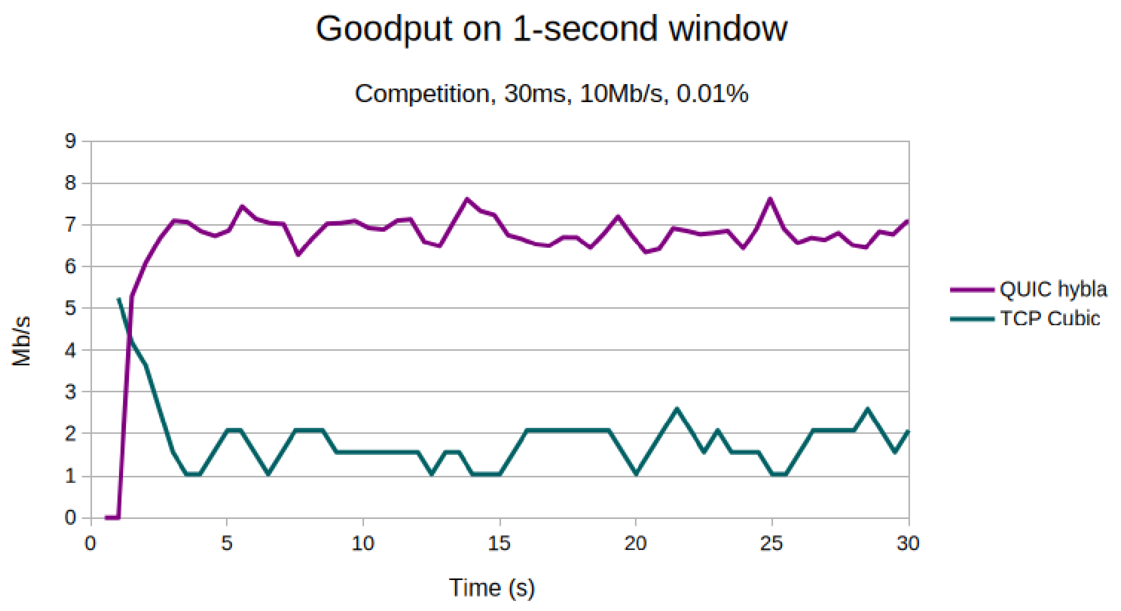

Figure 4.36: First 30 s of a connection that lasted 300 s, Competition between QUIC Hybla and TCP Cubic, RTT: 30 ms, bandwidth: 10 Mb/s, 0.01 % loss

4.4.2 GEO satellites

Let's now take into exam the GEO case. In this case, using iperf alone wouldn't work, since there is a delay that it's too big for TCP to work correctly. To resolve this limitation, it was used the pepsal mechanism [pepsal]. Basically, pepsal can be used in TCP connections that have to travel on a network that has at least one link characterized by a long propagation delay. To avoid having unexpected behavior from the TCP protocol, pepsal is added in those sections. In fact, it acts as a proxy by splitting the path in those areas, and then it uses TCP Hybla to communicate. In the sections that do not suffer high propagation delay, TCP Cubic is used. The source and the destination are completely unaware of the usage of pepsal, and, by adopting TCP Hybla, they are able to communicate without any workload added to them.

As a reference, the plot of the goodput when using TCP pepsal alone in a GEO environment is shown in Figure 4.37.

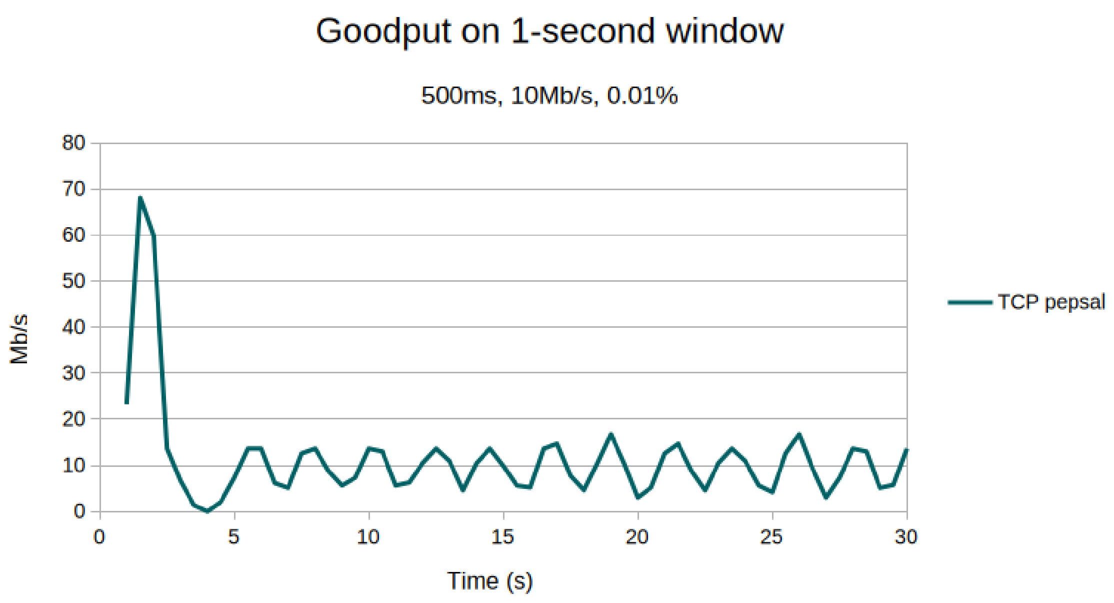

Figure 4.37: First 30 s of a connection that lasted 300 s, TCP pepsal, RTT: 500 ms, bandwidth: 10 Mb/s, 0.01 % loss

The plot of the congestion window of Hybla can be seen in Figure 4.38. Beside the initial 10 seconds, the value around the congestion window is fluctuating is slightly less than the experiment in the same situation but without competition [Figure 4.17]. Let's consider the plot of the goodput shown in Figure 4.39 to better understand the actual behavior in this case.

It seems clear that, besides the initial 15 seconds, the competition between QUIC Hybla and TCP pepsal is quite fair. The big spike at around 2.5 s in the TCP pepsal line can be blamed on the fact that probably the server was finally able to receive packets that were on flight and that provoked that sudden increase.

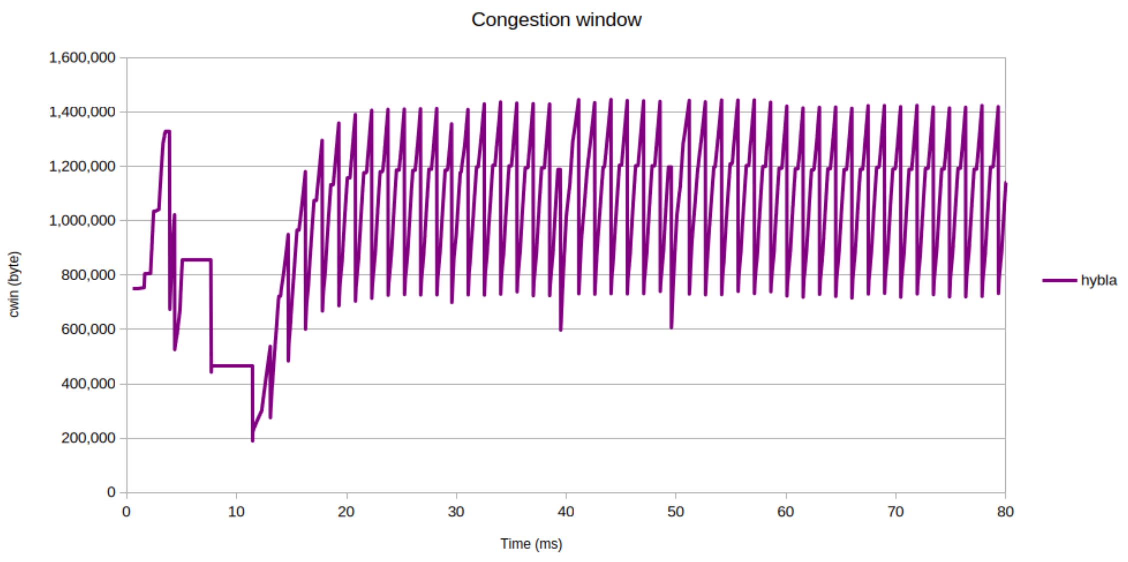

Figure 4.38: First 80 s of a connection that lasted 300 s, Competition between QUIC Hybla and TCP pepsal, RTT: 500 ms, bandwidth: 10 Mb/s, 0.01 % loss

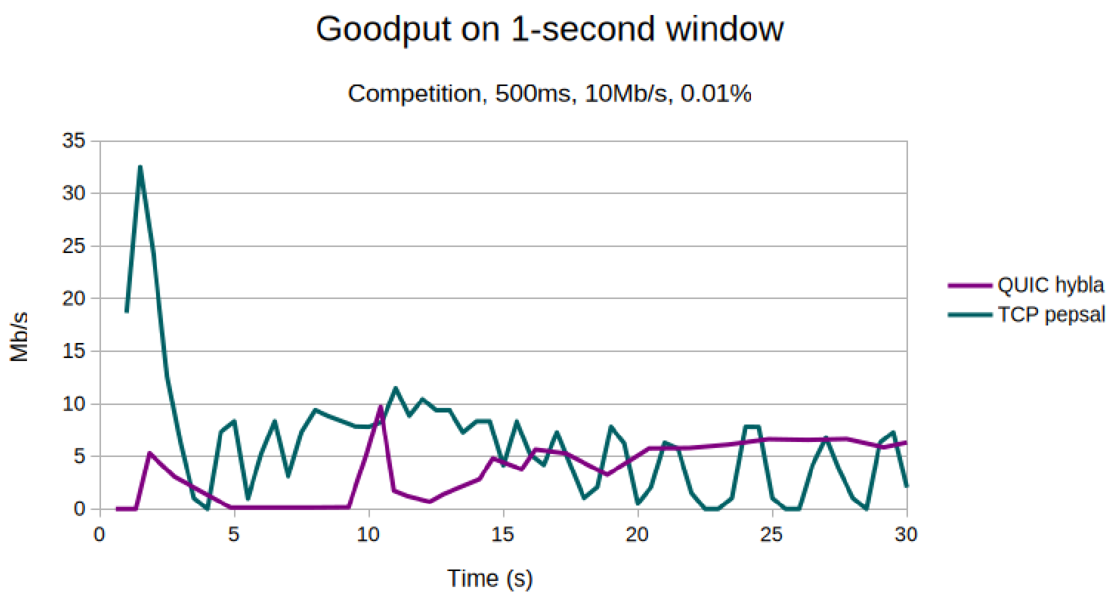

Figure 4.39: First 30 s of a connection that lasted 300 s, Competition between QUIC Hybla and TCP pepsal, RTT: 500 ms, bandwidth: 10 Mb/s, 0.01 % loss

Once again, the Jain fair index can give more insight about the actual friendliness of the connection. In this case, the average goodput for Hybla is $4.6566\,\text{Mb/s}$, while for TCP pepsal the average is $3.8636\,\text{Mb/s}$. This results in an index of 0.9914, which is a really good sign of friendliness.

Chapter 5

Performance evaluation of QUICCL in a few selected satellite networks

This section describes the tests performed in order to check the performance of the QUICCL developed by fellow student Moffa [Moffa_2025]. In particular, the testbed is the same as the previous 4.1.1, and the machines used to implement DTN nodes are VM1 and VM3 as sources, VM6 as destination. The configuration of the DTN hops includes three jumps: one between machine 1 and machine 6, one between machine 1 and machine 3, and, lastly, one between machine 3 and machine 6. In these scenarios, the destination (VM6) is a ground station, while the main source (VM1) is a satellite. VM3 represents another satellite.

As it was for the previous tests, a bash script is used in order to iterate automatically over the configurations. In particular, the latter are:

- **Data rate**: 100 Mb/s
- **Round-trip time**: 10 ms, 125 ms, 2 s2s
- **Packet error rate**: 10 %, 1 %, 0.1 %, 0.01 %

DTN communications are very easily disrupted by, for example, the movements of the planets or satellites, but also long propagation delays. In order to better test the performance in those cases, we added a system inside the testing script that automatically disrupts the connection for a specific period of time. The disruptions tested are:

- No disruptions
- **Off period**: 10 ms, 50 ms, 100 ms, 1 s, 10 s, 100 s, 200 s

This was achieved by adding an *iperf* rule on VM4 that acts as a firewall, discarding all packets originated from and directed to VM1. In addition to this, for the disruptions of 10, 100 and 200 seconds, a contact plan was added to the bundle protocol, basically letting it know that those disruption exist.

For the disruptions that are shorter, it doesn't make sense to inform the bundle protocol because even if it knew that one of those disruptions was coming, they are so short that it wouldn't even have time to react to them before the disruption is already over.

The disruptions tested are of two types:

- One where the transmission happens only between VM1 and VM6, meaning that during the disruptions nothing is delivered to the ground station.
- One where, during the disruptions, VM3 is visible by both VM1 and VM6, meaning that the machine 1 will send to the machine 3, making the latter responsible for the delivery of the bundles to machine 6.

As stated before, the latter is only available when the disruption is 10, 100 or 200 seconds.

or the operation of sending and receiving bundles, the program *dtnperf* is used [dtnperf]. This program is part of the DTNsuite [DTNsuite], which is a group of programs developed by the University of Bologna specifically for DTN communications. In particular, this tests were done by using a dtnperf server and monitor on the machine 6, and a dtnperf client on both sending machines (VM1 and MV3). The commands used are as follow:

- To launch a **server**:
```
dtnperf_vDTN2_vION_vUD3TN_vUNIBOBP --server
```
- To launch a **monitor**:
```
dtnperf_vDTN2_vION_vUD3TN_vUNIBOBP --monitor
```
- To launch a **client**:
```
dtnperf_vDTN2_vION_vUD3TN_vUNIBOBP --client -d ipn:6.2000 -T 300 -P1M -W32 --monitor ipn:6.1000
```

The client uses the time mode of dtnperf, sending bundles for a total of 300 seconds. These bundles have dimension of 1 Mb, and the window is set to 32. The value of the window represents the maximum amount of bundles that can be in flight at any given moment; this means that as soon as one bundle arrives at destination, the client can send another one (if the time passed is not higher than the duration set). The duration of the tests varies depending on the dimension of the disruption: when the disruption is shorter than 1 second, then the tests last 30 seconds, meanwhile, with disruptions of 10, 100 and 200 seconds, the tests last 5 minutes.

Another important step, that is necessary to set up the DTN communication, is the setting of the ranges, contacts and outducts. The range is used to specify the distance between the two machines; all the values for this experiments are written inside a bash script on the machines that automatically sets things up. The contacts are used to specify when a link between two peers is possible. In the experiments, only the contacts between VM3 and VM6, VM1 and VM3 are persistent; the contact between VM1 and VM6 is added and removed as necessary inside the testing script. For instance, when a longer disruption happens (10s, 100s and 200s), the VM1 cannot communicate directly with the ground station, but has to rely on the other satellite (VM3) to deliver the bundles. In this case the contact is fractured for a duration equal to the disruption and then it is brought back when the disruption ends. Lastly, outducts and inducts are what allow the peer to send and receive bundles for a specific underlying protocol. All the machines receive an induct the moment the bundle protocol agent is set up (through a bash script), meanwhile the outducts are added inside the testing script. This is done because a contact added after the outduct is activated will not be taken into consideration.

5.1 Single stream

QUICCL provides two different modes of operating when deciding the number of streams: one where only one stream is opened and all the bundles will be sent on it, and another where every bundle opens a new stream. This section describes the former.

5.1.1 Round-trip time: 10 ms

Let us begin by considering the case in which the RTT is set to 10 ms, the loss rate is set to 0.01 %, and there is no disruption. As stated before, the duration of the test is 30 s. Figure 5.1 shows a plot where every point represents the moment a bundle is generated (in blue) and then received (orange). This means that the space between a blue dot and an orange dot laying on the same horizontal line is the time that has passed before the bundle reached its destination. This plot is what happens in an almost perfect situation: the delay is not big, the loss rate is low, and there is no disruption.

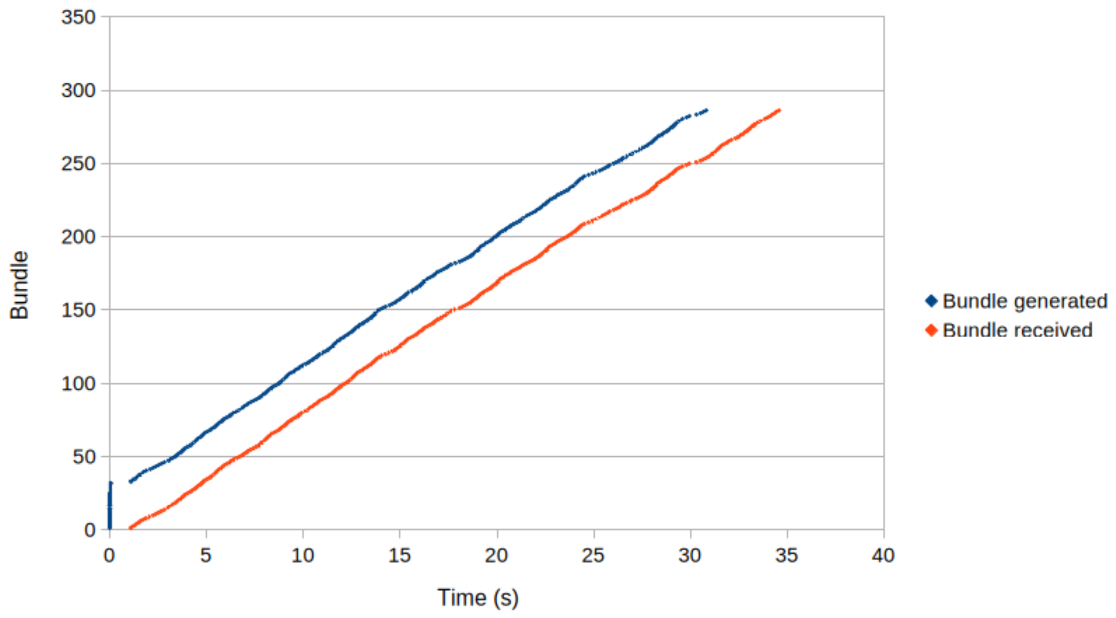

Figure 5.1: Transmission of $1\,\mathrm{MB}$ bundles over $30\,\mathrm{s}$, bandwidth $100\,\mathrm{Mb/s}$, $10\,\mathrm{ms}$ round-trip time, $0.01\,\%$ packet error rate, $W = 32$

The same experiment but with a disruption of $1\,\mathrm{s}$ is visible in Figure 5.2. It is clear that from 14 seconds to 16 seconds the client is stopped from sending bundles.

Since the disruption is too short, the traffic is not redirected to the VM3. In order to analyze such case, another test is taken into consideration. This time, the difference are the duration of the test (300 seconds) and the duration of the disruption (100 seconds). The most interesting detail about the results of this test is shown in Figure 5.3.

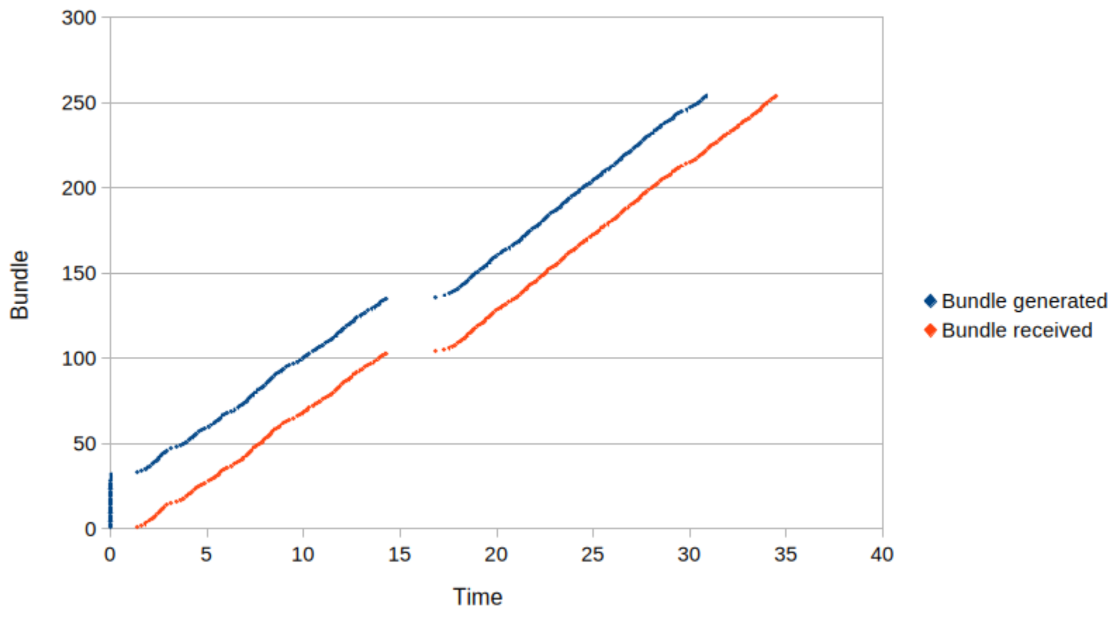

Figure 5.2: Transmission of 1 MB bundles over 30 s, bandwidth 100 Mb/s, 10 ms round-trip time, 0.01 % packet error rate, 1 s disruption, $W = 32$

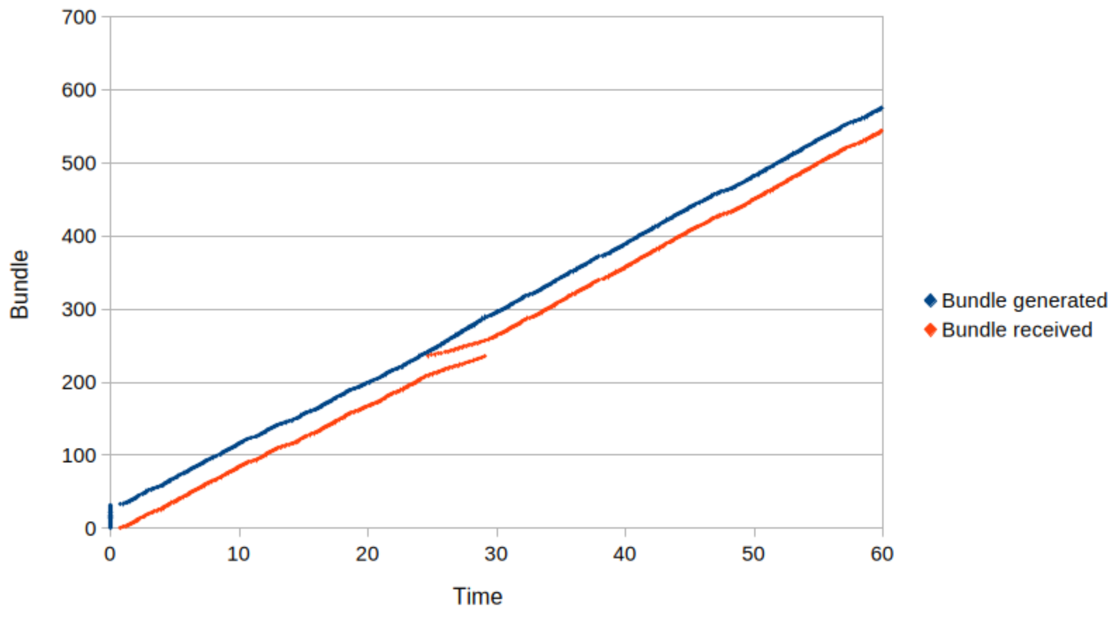

Figure 5.3: Detail of the first 60 s of a transmission of 1 MB bundles over 300s, bandwidth 100 Mb/s, 10 ms round-trip time, 0.01 % packet error rate, 100 s disruption, $W = 32$

This plot only represents the first minute of the test, and the disruption begins at 30 seconds. It is very clear that the bundle protocol agent starts using the alternative path way before the disruption happens. This is because, from the point of view of VM1, the machine 3 is always visible up until the end of the disruption. Unibo-BP's internal algorithms, based on the contact's transmission rate, ensure that the alternative path is taken in advance to guarantee that the bundles actually reach the recipient. If the original path was terminated exactly at the closure of the contact, the last bundles sent would never be received.

The reason why the orange dots are very close to the blue one when the bundle protocol agent switches path is because the dtnperf client is limited by a window size of 32 bundles. For instance, at the beginning the client sends out a burst of 32 bundles, and then waits for the reception of acknowledgements. Once the ACK for the first bundle is received (there is only one stream, so the bundles are delivered in order), the client is free to send another bundle. This proceeds for the whole duration of the connection. When the switch takes place, a new QUICCL instance is opened, from the VM3 to the VM6, meaning that another stream is created. This stream does not have a queue yet, so the acknowledgement of the first bundle is received almost immediately. The reason why the shape of the orange dots is slightly curved instead of a straight line is because the client cannot send a burst of 32 bundles, unlike the start of the connection (blue vertical line at time 0). It is not shown in this document, but the behavior of the longer disruptions, in the case in which VM3 does not operate, is exactly the same as the one with shorter disruptions (the only difference is the size of the void between the dots).

5.1.2 Round-trip time: 125 ms

The tests with a round trip time of 125 ms produced similar results in respect to the "shape" of the graph. The only difference is the density of bundles generated, result that is to be expected since higher propagation delays are bound to further degrade performance. An example of this is Figure 5.4.

The addition of packet error rate is also a factor that contributes to the downgrade of the behavior. For instance, the same test shown in Figure 5.4, but with a loss rate of 1 % produces the plot shown in Figure 5.5. The disruption happens at around 14 seconds. It is interesting to notice how the presence of even the 1 % of loss rate change the final result of the experiment. It is simple to notice that the acknowledgement of the last bundle arrived almost 30 seconds later compared to the experiment with low packet error rate, but also the amount of dots is incredibly low.

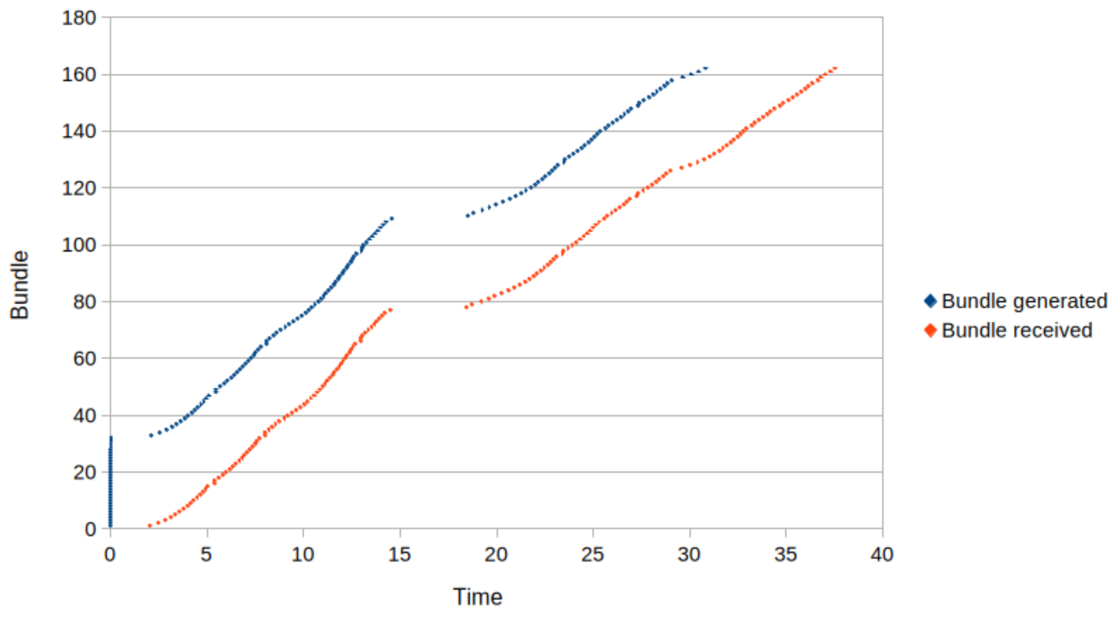

Figure 5.4: Transmission of 1 MB bundles over 30 s, bandwidth 100 Mb/s, 125 ms round-trip time, 0.01 % packet error rate, 1 s disruption, $W = 32$

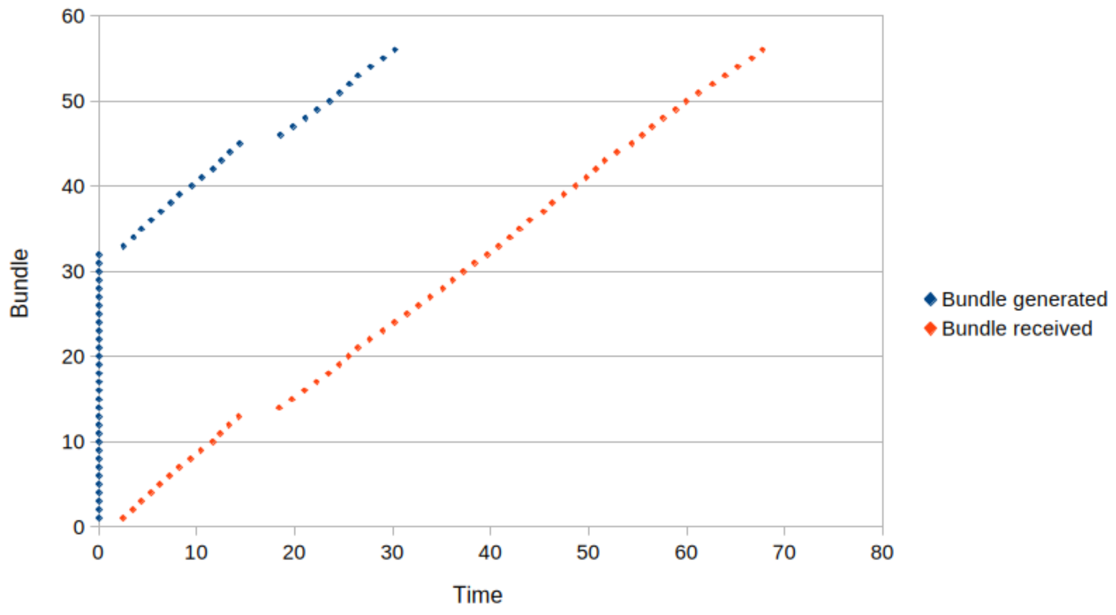

Figure 5.5: Transmission of 1 MB bundles over 30 s, bandwidth 100 Mb/s, 125 ms round-trip time, 1 % packet error rate, 1 s disruption, $W = 32$

5.1.3 Round-trip time: 2 s

For what concerns tests with a round trip time of 2 seconds, the graph in Figure 5.6 shows the result of a test run over the duration of 30 seconds, with a disruption of 1 s and a packet error rate of $0.01\,\%$.

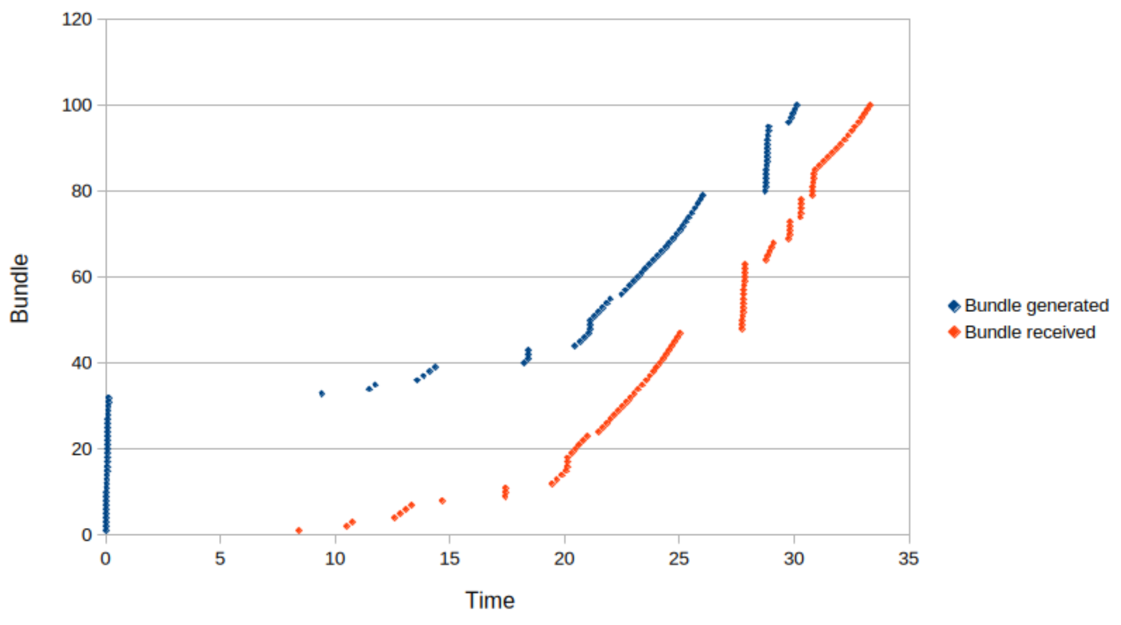

Figure 5.6: Transmission of 1 MB bundles over 30 s, bandwidth $100\,\mathrm{Mb/s}$, 2 s round-trip time, $0.01\,\%$ packet error rate, 1 s disruption, $W = 32$

In this graph, the effects of a longer round-trip time are really visible. The disruption happened around 15 seconds. The reason why it takes almost 8 seconds for the first acknowledgement to come back to VM1 is firstly due to the propagation delay, but also because having bundles that consists of 1MB and a window set in dtnperf of 32, the bandwidth of $100\,\mathrm{Mb/s}$ is filled extremely quickly, creating queues that introduce more delay.

To have an idea of the goodput of these experiments, one could simply look at the amount of bundles that were generated (blue dots). Since the dimension of the bundle is always the same (1 MB), and the duration of the tests is fixed, calculating the average goodput of the connection is trivial.

5.2 Multiple streams

Lastly, as a reference, Figure 5.7 shows the plot of an experiment conducted using the multi-stream option offered in QUICCL. In this case, a QUIC stream is created for each individual stream, meaning that the bundles from a single burst are sent in parallel and share the same bandwidth.

The reason why the plot looks like this is because the bundles are sent on different streams, through a burst, and, in the same fashion, the acknowledgement behave the same. This means that the client is able to generate bundles up to the maximum of the minimum between the window size or the bandwidth, and then has to wait for the acknowledgement to come back before sending again.

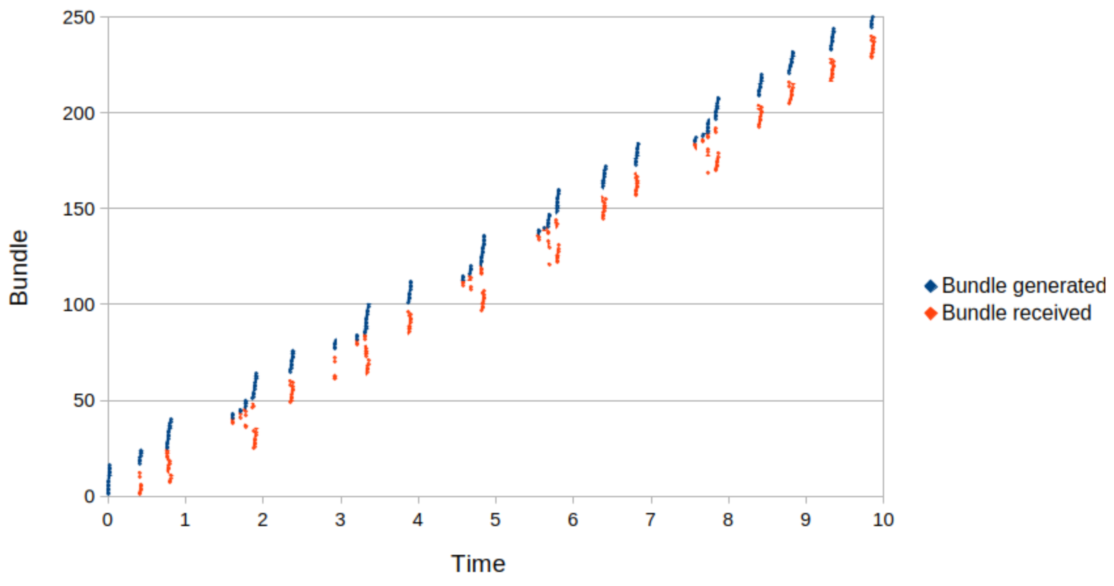

Figure 5.7: Transmission of 1 MB bundles over 10 s using multi-stream option, $W = 32$

Chapter 6

Conclusions

The objective of this thesis was to carry out tests and then examine the performance of the products of the two companion theses of fellow students Cavallotti Valentino and Mattia Moffa, and, in order to do so, to develop the tools necessary for evaluating them. As part of a broader research project developed at the German Aerospace Center (DLR) in collaboration with the University of Bologna, the common interest was to investigate the potential of QUIC in satellite and interplanetary networks.

Firstly, through a careful analysis and thorough comprehension of the QUIC specification and its available implementation, Picoquic was chosen as the best candidate for this project. Following an in-depth inspection of the source code, the first extension to the testing program provided by Picoquic was developed: the possibility to utilize a time mode for the test scenarios. The use of this new feature led to the need of a more detailed logging system, which was then added as part of the program itself. These additions were crucial for the correct unfolding of the tests.

After the developing phase, the focus shifted towards setting up a testbed suitable for the ambitions of this project. In particular, a Virtualbricks project comprised of 5 virtual machines was set up and experimented on to assure that all the components worked correctly.

Lastly, after setting up a bash script that automatically iterates on the different configurations, the tests were actually set in motion. The following step was to collect all the data and organize it in graphs, in order to better visualize the results. This last process was also automated by use of a Python script.

In conclusion, the results presented in the last two chapter of this document confirm that the project was indeed successful. The implementation of Hybla in QUIC proved to be adequate when used in networks characterized by high delays, surpassing NewReno and reaching a similar value to the one of Cubic and BBR. The evaluations also show that QUICCL was able to operate correctly under conditions that are typical of satellite and interplanetary scenarios, while maintaining a high level of performance.

Bibliography

[BBR_draft] Neal Cardwell, Ian Swett, and Joseph Beshay, *BBR Congestion Control*, Internet-Draft draft-ietf-ccwg-bbr-02, Work in Progress, Internet Engineering Task Force, Feb. 2025, 78 pp., URL: `https://datatracker.ietf.org/doc/draft-ietf-ccwg-bbr/02/`.

[Caini_2004] Carlo Caini and Rosario 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`, eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/sat.799`, URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/sat.799`.

[Cavallotti_2025] Valentino Cavallotti, "Design and implementation of a QUIC congestion control designed after TCP Hybla", Master's thesis, University of Bologna, Mar. 2025.

[Chrome] *Link to Chrome release notes*, URL: `https://chromiumcodereview.appspot.com/11125002/`.

[CNRL] *Link to CNRL website*, URL: `http://cnrl.deis.unibo.it/VB_projects.php`.

[dtnperf] *Link to dtnperf GitLab website*, URL: `https://gitlab.com/dtnsuite/dtnperf`.

[DTNsuite] *Link to DTNsuite GitLab website*, URL: `https://gitlab.com/dtnsuite/`.

[Farrell_2011] Carlo Caini et al., "Delay- and Disruption-Tolerant Networking (DTN): An Alternative Solution for Future Satellite Networking Applications", in: *Proceedings of the IEEE* 99 (Nov. 2011), pp. 1980–1997, DOI: 10.1109/JPROC.2011.2158378.

[Fast_TCP] X. Wei David et al., “Fast TCP: motivation, architecture, algorithms, performance”, in: (2006), URL: https://web.archive.org/web/20060906104225/http://netlab.caltech.edu/pub/papers/FAST-ToN-final-060209.pdf.

[Firefox] *Link to Firefox release notes*, URL: https://hacks.mozilla.org/2021/04/quic-and-http-3-support-now-in-firefox-nightly-and-beta/.

[Huitema] *Link to Christian Huitema’s blog*, URL: https://www.privateoctopus.com/blog.html.

[ION] *Link to ION source code*, URL: https://sourceforge.net/projects/ion-dtn/.

[Iperf] *Link to iperf website*, URL: https://iperf.fr/.

[Moffa_2025] Mattia Moffa, “Design and implementation of a DTN Convergence Layer Adapter based on the QUIC protocol”, Master’s thesis, University of Bologna, Mar. 2025.

[openSSL] *Link to openSSL official website*, URL: https://www.openssl.org/.

[pepsal] C. Caini, R. Firrincieli, and D. Lacamera, “PEPsal: a Performance Enhancing Proxy designed for TCP satellite connections”, in: *2006 IEEE 63rd Vehicular Technology Conference*, vol. 6, May 2006, pp. 2607–2611, DOI: 10.1109/VETECS.2006.1683339.

[Picoquic] *Picoquic source code*, URL: https://imgur.com/a/BZDW42Q/.

[Picoquic_MAC] *Picoquic fork source code*, URL: https://github.com/MAC-Projects/picoquic/.

[Picoquicdemo] *Link to picoquicdemo website*, URL: https://www.privateoctopus.com/picoquic/quicperf.html/.

[Picotls] *Picotls source code*, URL: https://github.com/h2o/picotls/.

[Qlog_draft] Robin Marx et al., *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/.

[quic-perf_draft] Nick Banks, *QUIC Performance*, Internet-Draft draft-banks-quic-performance-00, Work in Progress, Internet Engineering Task Force, Dec. 2020, 8 pp., URL: `https://datatracker.ietf.org/doc/draft-banks-quic-performance/00/`.

[qvis] *Link to QLOG visualizer website*, URL: `https://qvis.quictools.info/`.

[RFC4838] Leigh Torgerson et al., *Delay-Tolerant Networking Architecture*, RFC 4838, Apr. 2007, DOI: `10.17487/RFC4838`, URL: `https://www.rfc-editor.org/info/rfc4838`.

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

[RFC6582] Andrei Gurtov et al., *The NewReno Modification to TCP's Fast Recovery Algorithm*, RFC 6582, Apr. 2012, DOI: `10.17487/RFC6582`, URL: `https://www.rfc-editor.org/info/rfc6582`.

[RFC7301] Stephan Friedl et al., *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`.

[RFC8446] Eric Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*, RFC 8446, Aug. 2018, DOI: `10.17487/RFC8446`, URL: `https://www.rfc-editor.org/info/rfc8446`.

[RFC9000] Jana Iyengar and Martin 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] Martin Thomson and Sean Turner, *Using TLS to Secure QUIC*, RFC 9001, May 2021, DOI: `10.17487/RFC9001`, URL: `https://www.rfc-editor.org/info/rfc9001`.

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

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

[RFC9293] Wesley Eddy, *Transmission Control Protocol (TCP)*, RFC 9293, Aug. 2022, DOI: `10.17487/RFC9293`, URL: `https://www.rfc-editor.org/info/rfc9293`.

[RFC9438] Lisong Xu et al., *CUBIC for Fast and Long-Distance Networks*, RFC 9438, Aug. 2023, DOI: `10.17487/RFC9438`, URL: `https://www.rfc-editor.org/info/rfc9438`.

[Safari24] *Link to Safari version 24 release notes*, URL: `https://developer.apple.com/documentation/safari-release-notes/safari-14-release-notes/`.

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

[Virtualbricks] Pietrofrancesco Apollonio et al., "Virtualbricks for DTN Satellite Communications Research and Education", in: *Personal Satellite Services. Next-Generation Satellite Networking and Communication Systems*, ed. by Igor Bisio, Cham: Springer International Publishing, 2016, pp. 76–88, ISBN: 978-3-319-47081-8.

[Warthman_2015] Forrest Warthman and Warthman Associates, "Delay- and Disruption-Tolerant Networks (DTNs) - A Tutorial", in: (Sept. 2015), URL: `https://www.nasa.gov/wp-content/uploads/2023/09/dtn-tutorial-v3.2-0.pdf`.

Acknowledgements

I would like to thank all the people who have stood by my side over these past years; without them, I wouldn't be where I am today.

First and foremost, I thank my parents for their unwavering support and for always believing in my dreams.

A special thank you goes to Tommaso for always being by my side, supporting me, and accompanying me every day — his support is my greatest strength.

A heartfelt thank you to my friends, especially Mattia and Valentino, who have made these years of study lighter and more manageable.

Lastly, I would like to my gratitude to my co-supervisor, Tomaso de Cola, for his constant support, guidance and invaluable assistance throughout this project.