ALMA MATER STUDIORUM
UNIVERSITY OF BOLOGNA

Department of Computer Science and Engineering

Master's Degree in Computer Engineering

MASTER'S THESIS

in

Protocols and Architectures for Space Networks M

Design and implementation of a QUIC congestion control designed after TCP Hybla

Candidate

Valentino Cavallotti

Supervisor

Prof. **Carlo Caini**

Co-supervisor

Dott. Ing. **Tomaso de Cola**

Academic Year 2023/2024

Abstract

The Internet architecture follows a multi-layered structure, where each layer is associated with specific communication protocols – standardized sets of rules that determine how information (application data and/or control data) is sent, received, and recovered.

Standard internet protocols provide functionality that is well-suited to Earth-based networks. Networks involving satellite links, on the other hand, may face additional challenges, such as higher propagation times, higher loss rates due to attenuation and impairments, and intermittent links due to the relative movement of nodes.

Satellite networks use satellites to provide connectivity over vast geographic areas, and communicate with ground stations to control and relay transmission between satellites and users. They can be differentiated based on the distance from Earth of the satellite nodes involved:

- GEO (Geostationary Earth Orbit) satellites cover large portions of Earth thanks to their distance, but have high latency, with round-trip times of over 600 ms. They orbit at an altitude of 35 786 km, which allows them to match the Earth's rotational period; their position is fixed relative to the planet's surface, making GEO links typically available.
- LEO (Low Earth Orbit) satellites are closer to Earth, and have lower latency but reduced coverage. The lower orbit forces them to move at a higher speed than the Earth's rotation, making LEO links intermittent.

QUIC [RFC9000] is a transport-level protocol originally developed by Google in 2012 as a replacement to TCP in HTTP communications. This thesis, alongside two companion theses [Moffa_2025] [Andreetti_2025], is part of a project with the goal of evaluating the performance of the QUIC protocol achievable in GEO and LEO satellite networks. This project has been developed at the Institute of Communication and Navigation of the German Aerospace Center (Deutsches Zentrum für Luft- und Raumfahrt, DLR), in collaboration with the University of Bologna.

For QUIC to operate effectively in such scenarios, it is critical to configure it in a way that addresses the unique challenges of satellite networks. An important part of this configuration

work consists in equipping the protocol with an appropriate congestion control algorithm.

Congestion control is a critical aspect of networking, by which endpoints regulate the amount of data they send into the network, dynamically adapting their output based on losses and latency. Typical congestion control algorithms are designed to handle Earth-based communications, and may face problems in satellite networks.

The QUIC RFC does not impose the use of a specific congestion control strategy, though a NewReno variant is described in one of the QUIC RFCs [RFC9002] - the only constraint is that a standard QUIC receiver should be compatible with the chosen algorithm.

Hybla [Caini_2004] is a congestion control algorithm originally proposed for TCP in a 2004 paper by Carlo Caini and Rosario Firrincieli. It is specifically designed to handle the long delays typical of GEO satellite links, which makes it of great interest for the project. The object of this thesis is thus the design and implementation of a version of Hybla for QUIC.

This work relies on an existing implementation of the QUIC protocol, Picoquic – an open-source, portable, minimalist implementation, written in C, originally created – and primarily maintained, to this day – by Christian Huitema [Picoquic].

Contents

Chapter 1 The QUIC protocol

1.1 Overview

QUIC is a general-purpose, connection-oriented, reliable transport protocol, designed to replace TCP in many applications, primarily in the context of HTTP3. The original QUIC specification was first drafted in 2012 by Jim Roskind, at Google. The protocol was first submitted for standardization as an Internet-Draft in 2015, and in May 2021 it was formally defined by the IETF in a series of companion RFCs: [RFC8999], [RFC9000], [RFC9001], [RFC9002].

RFC 8999 delineates the version-independent properties of the protocol, that can always be relied on regardless of the QUIC version in use. RFC 9000 defines the core features of the protocol. RFC 9001 describes how Transport Layer Security (TLS) is used with QUIC; finally, RFC 9002 describes QUIC mechanisms for loss detection and congestion control, and as such is of particular interest for this thesis.

The protocol has been refined further in the years following the initial RFCs, with additional drafts and RFCs addressing updates and extensions. Notable examples include RFC 9369, which specifies version 2 of QUIC (with minor differences compared to the original) [RFC9369], and RFC 9221, which describes an extension for the transmission of unreliable datagrams [RFC9221].

QUIC provides features such as flow-controlled streams, low-latency connection establishment, network path migration, and more. It includes security measures to ensure confidentiality, integrity, and availability.

QUIC endpoints exchange units named *packets*, which are encapsulated within UDP datagrams; a UDP datagram can contain one or more QUIC packets.

There are six types of packets (Version Negotiation, Initial, Handshake, 0-RTT, 1-RTT, and Retry), which differ in the structure and content of their header. 1-RTT is the main packet type for application data; 0-RTT packets are used to send application data during the connection handshake (when possible), and the remaining packet types are used to exchange specific control information. Aside for Version Negotiation and Retry packets, all packets are numbered, which facilitates their management.

Packets contain one or more *frames*, units that may carry either control information or application data. There are many frame types, such as `STREAM`, `RESET_STREAM`, `STOP_SENDING`, `ACK`, `PADDING`, `CRYPTO`, `CONNECTION_CLOSE`, and more, each one with its own set of internal fields.

Applications engaged in a QUIC connection exchange information through *streams*. Streams are abstractions provided to endpoints that represent ordered sequences of bytes. They can be either unidirectional (only the endpoint initiating the stream can use it to send data) or bidirectional (both endpoints can send data).

An arbitrary number of streams can operate concurrently, and they can carry arbitrary amounts of data (while under flow and congestion control constraints). Stream creation and the amount of data sent are managed using a credit-based scheme.

Streams are multiplexed within packets, i.e a single packet may contain frames belonging to different streams.

Packets are confirmed by `ACK` frames, as TCP segments are by TCP `ACK`s, though there are a few significant differences between the two mechanisms.

- Firstly, QUIC `ACK`s refer to packet numbers rather than to the bytes transported by the packets.
- Secondly, though all packets must be confirmed, not all packets trigger an acknowledgment. More precisely, the acknowledgment logic is affected by the frame types within a packet: some frames types are "`ACK`-eliciting", others are not. An "`ACK`-eliciting packet" is one that contains any `ACK`-eliciting frames. Only `ACK`-eliciting packets require an acknowledgment to be sent back immediately upon reception, while other packets are acknowledged alongside `ACK`-eliciting packets.

`ACK` frames are designed to make loss recovery fast, and reduce the number of retransmissions. Each `ACK` can acknowledge an arbitrary number of packets, either contiguous or with gaps in between. This is done through a field of the `ACK` frame that contains a list of ranges of packet numbers, where each range of the list indicates a contiguous sequence of packets acknowledged. This mechanism is reminiscent of the SACK option in TCP, but it operates on intervals of packet numbers instead of byte intervals, and does not have SACK's three-interval limit.

It is worth noting that packets determined lost are not retransmitted whole, and neither are frames. Instead, it is the information carried in the lost frames that is sent again, in new frames and packets, as needed.

As QUIC is a connection-oriented protocol, it requires a phase of connection establishment. The QUIC handshake is designed to establish a connection quickly and permit the exchange of application data as soon as possible. QUIC assumes transport-layer security to be a necessary feature, so its handshake combines the negotiation of cryptographic (TLS) and transport parameters. An option named "0-RTT", or "Early data", allows the sender to begin sending data during the handshake, fully eliminating the already-reduced initial wait. However, using this option requires prior communication or configuration.

QUIC connections allow for connection migration: connections are not bound to a specific transport-level path, and can be transferred to a new one if needed, through a mechanism based on connection IDs.

To avoid network congestion, QUIC relies on congestion control algorithms, which work analogously to those of TCP: given a connection — or, more specifically, a transport-level path between two endpoints, identified by a four-value tuple of two IPs and two ports — the congestion control algorithm regularly updates a "congestion window" value, which acts as a limit to the amount of outgoing data on all streams on that path, cumulatively.

1.2 Streams

Streams are lightweight channels of ordered bytes within a QUIC connection. They can be unidirectional or bidirectional. A QUIC connection can have many streams, which can operate concurrently, and can be created by either endpoint. While data on a single stream are ordered, there is no established ordering between bytes belonging to different streams: this makes streams very efficient for the transmission of a multitude of individual elements (e.g. the elements that make up a web page, which may arrive in any order), but less effective when it comes to order-sensitive data, such as that of a multimedia flow.

Each stream is identified within its connection by an integer, the stream ID. The least significant bit of the stream ID indicates whether the stream is client-initiated (`0`) or server-initiated (`1`); the second least significant bit indicates whether the stream is unidirectional (`0`) or bidirectional (`1`).

Application data are transmitted on streams, encapsulated in frames of type `STREAM`. The act of sending a `STREAM` frame in a QUIC packet can implicitly create a stream, meaning stream creation has no overhead. Stream management operations (e.g. ending streams, canceling streams, managing stream flow control) are all designed to reduce overhead to a minimum.

A `STREAM` frame contains the following fields:

- Type: a value ranging from `0x08` to `0x0F`, of which only the three least significant bits matter. These bits are referred to as `OFF` (indicates whether an Offset field is present), `LEN` (indicates whether a Length field is present), and `FIN` (indicates whether this `STREAM` frame marks the end of the corresponding stream).
- Stream ID of the corresponding stream.
- Data length; if not present, the stream data is assumed to extend all the way to the end of the packet.
- Offset in bytes - of the frame's data, relatively to the data sent on the stream. Not present in a frame that marks the very start of a stream.
- Stream data.

The receiving endpoint uses the Stream ID and Offset fields to sort received data.

Applications can assign different priorities to streams, which determines how the QUIC implementation allocates available resources to them. Appropriate stream prioritization can have a significant effect on application performance.

A stream between two endpoints can be seen as having two “ends” (referred to as “parts”), one at each endpoint. In a unidirectional stream, one part is just for sending, and the other is for receiving; in a bidirectional stream, both parts have both purposes.

An application that owns the sending part of a stream can:

- Write data on the stream,

- End the stream ("clean" termination, via a `STREAM` frame with the `FIN` bit set),
- Reset the stream ("abrupt" termination, via a `RESET_STREAM` frame).

On the other hand, using the receiving part of a stream, an application can:

- Read stream data,
- Abort reading of the stream and request closure.

An application can also request to be notified by the QUIC protocol in a variety of circumstances: when a stream is opened or reset by its peer, when its peer aborts reading on a stream, when new data is available on a stream, and when data can or cannot be written to a stream due to flow control or congestion control constraints.

Both the sending and receiving parts of a stream transition between various states as they operate. The behavior of sending and receiving parts over time can be represented by a state machine (or two at once, in the case of bidirectional streams).

1.2.1 Sending states

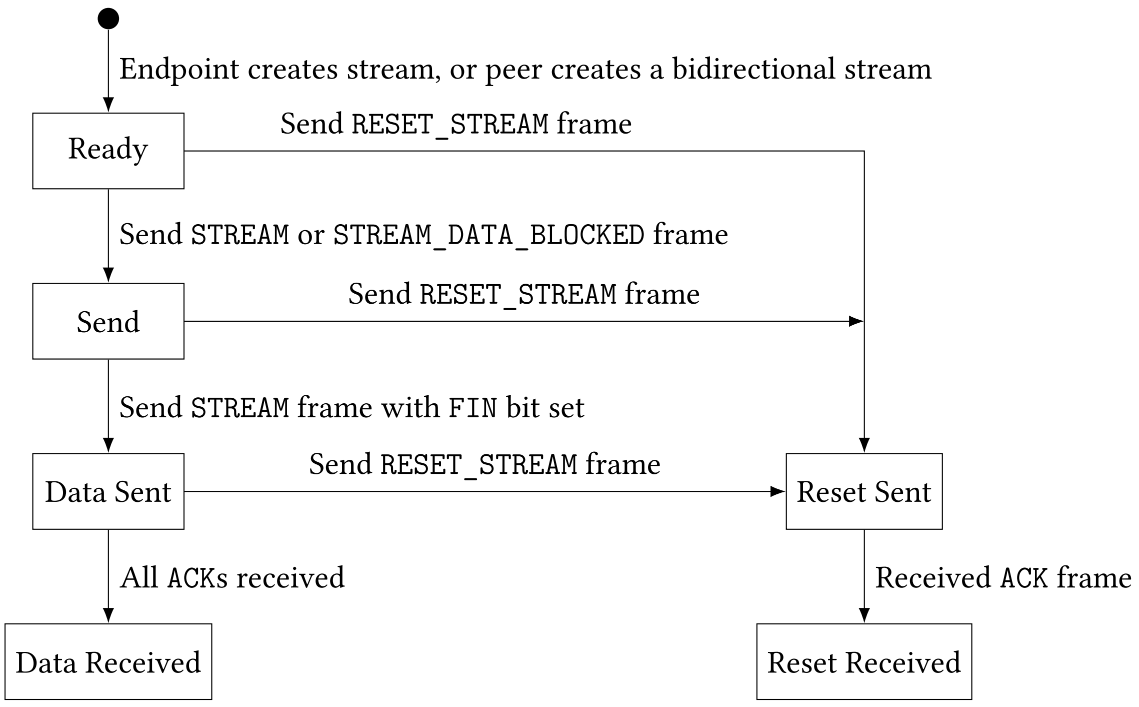

Figure 1.1: State machine of the behavior of the sending part of a stream.

When an endpoint creates a stream, or, alternatively, when its peer creates a bidirectional stream, the endpoint acquires the sending part of that stream.

The sending part begins in the “Ready” state. A sending part in this state can accept data from the application. As soon as the first `STREAM` (or, alternatively, `STREAM_DATA_BLOCKED`) frame is sent, the part enters the “Send” state (`STREAM_DATA_BLOCKED` is a type of frame generated by the sender when it is blocked from sending data by stream flow control limits).

An endpoint in the “Send” state transmits and/or retransmits data in `STREAM` frames.

When the application has sent all stream data, the sending part sets the `FIN` bit on the last `STREAM` frame sent, and enters the “Data Sent” state. In this state, only retransmissions can occur.

Once all stream data has been acknowledged, the part enters the “Data Received” state, and the sending part closes.

During the “Ready”, “Send”, or “Data Sent” states, it may happen that the application decides to cancel data transmission, or that the endpoint receives a `STOP_SENDING` frame from the peer. In both cases, the endpoint will send a `RESET_STREAM` frame, and enter the “Reset Sent” state.

- `RESET_STREAM` is a type of frame generated by a sender endpoint in order to abruptly terminate a stream. No transmission or retransmission of `STREAM` frames will occur after a `RESET_STREAM` has been sent.
- `STOP_SENDING` is a frame sent by an endpoint whose application has indicated that transmission on a specific stream should cease, and any incoming data will be discarded on receipt.

When the packet containing the `RESET_STREAM` frame is acknowledged, the endpoint enters the “Reset Received” state, and the sending part terminates.

1.2.2 Receiving states

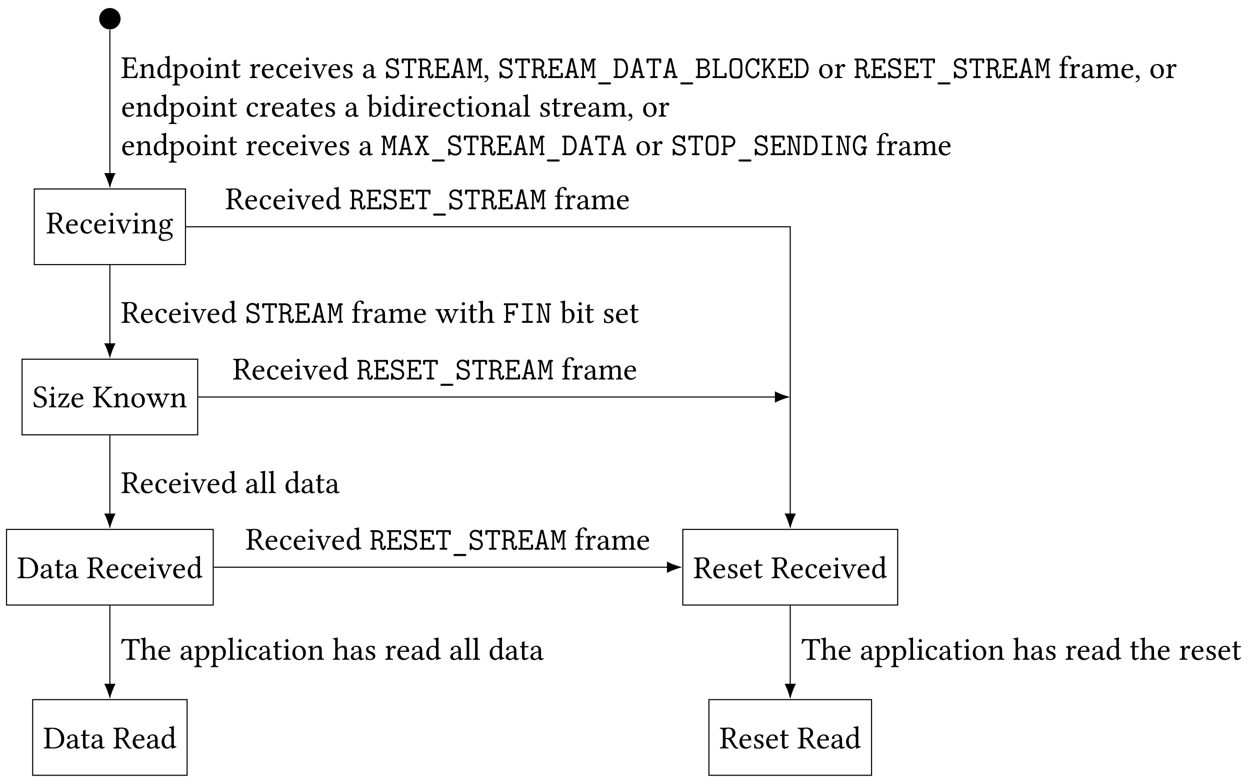
Endpoint receives a STREAM, STREAM_DATA_BLOCKED or RESET_STREAM frame, or
endpoint creates a bidirectional stream, or
endpoint receives a MAX_STREAM_DATA or STOP_SENDING frame
Receiving
Received RESET_STREAM frame
Received STREAM frame with FIN bit set
Size Known
Received RESET_STREAM frame
Received 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 1.2: State machine of the behavior of the receiving part of a stream.

An endpoint will automatically create the receiving part for a stream when:

- It creates a bidirectional stream,
- It receives the first `STREAM` (or `STREAM_DATA_BLOCKED`, or `RESET_STREAM`) frame on a new stream created by the peer.
- It receives a `STREAM`, `STREAM_DATA_BLOCKED`, `RESET_STREAM`, `MAX_STREAM_DATA`, or `STOP_SENDING` frame associated to a bidirectional stream created by the peer. In this case, the endpoint will create both a sending part and receiving part for that stream.
  - `MAX_STREAM_DATA` is a flow control frame that informs the peer about the maximum amount of data that can be sent on a specific stream.

Frames such as `MAX_STREAM_DATA` or `STOP_SENDING` may arrive before any `STREAM` frame if packets happen to be lost or shuffled.

A receiving part begins in the “Receiving” state. In this state, the endpoint receives `STREAM` frames carrying stream data (and STREAM_DATA_BLOCKED frames, when the sender gets blocked by flow control). Stream data received are buffered in a stream-specific buffer, and progressively consumed by the application. The endpoint sends out MAX_STREAM_DATA frames to inform the sender of new available buffer space for a stream, thus allowing more data to be sent on it.

When the endpoint receives a STREAM frame with the FIN bit set, the receiving part enters the "Size Known" state. In this state, MAX_STREAM_DATA frames are no longer sent to the sender, since no new data will be received, and only retransmissions can occur.

Once all stream data have been received, the receiving part enters the "Data Received" state; then, when the application has read all data, the part enters the "Data Read" terminal state.

Alternatively, the endpoint may at any time receive a RESET_STREAM frame, putting the receiving part in the "Reset Received" state. When the application becomes aware of the fact the stream was reset, the receiving part moves to the "Reset Read" state, and the receiving part terminates.

1.3 Flow control

The aim of flow control is to avoid losses on the receiver's side. The receiver provides information regarding its available buffer capacity, which the sender uses to regulate the amount of data to send.

QUIC flow control operates on two levels:

- Stream flow control: limits the amount of bytes of data that can be sent on each stream, so that a single stream does not consume the entire receive buffer of the connection.
- Connection flow control: limits the amount of bytes of data that can be sent on all streams cumulatively, in order to prevent a sender from exceeding the receiver's total buffer capacity.

Initial stream and connection flow control limits are set as transport parameters during the initial handshake. After that, the receiver will periodically send MAX_STREAM_DATA and MAX_DATA frames to provide the sender with updated limits. The former is for stream flow control, and the latter is for the connection.

A sender that has sent data up to the provided limit will periodically send `STREAM_DATA_BLOCKED` or `DATA_BLOCKED` frames to inform the receiver that it is being blocked due to flow control.

The receiving endpoint also controls the maximum total number of streams that its peer can create. The initial limit is set as a transport parameter during the handshake, and subsequent limits are advertised using `MAX_STREAMS` frames.

1.4 Connections

A QUIC connection begins with a handshake phase. The QUIC handshake negotiates both transport and cryptographic parameters simultaneously, which minimizes connection establishment latency. This is in contrast to TCP, where the handshake only negotiates transport parameters initially, and a separate TLS handshake handles encryption setup immediately afterwards.

1.4.1 Handshake

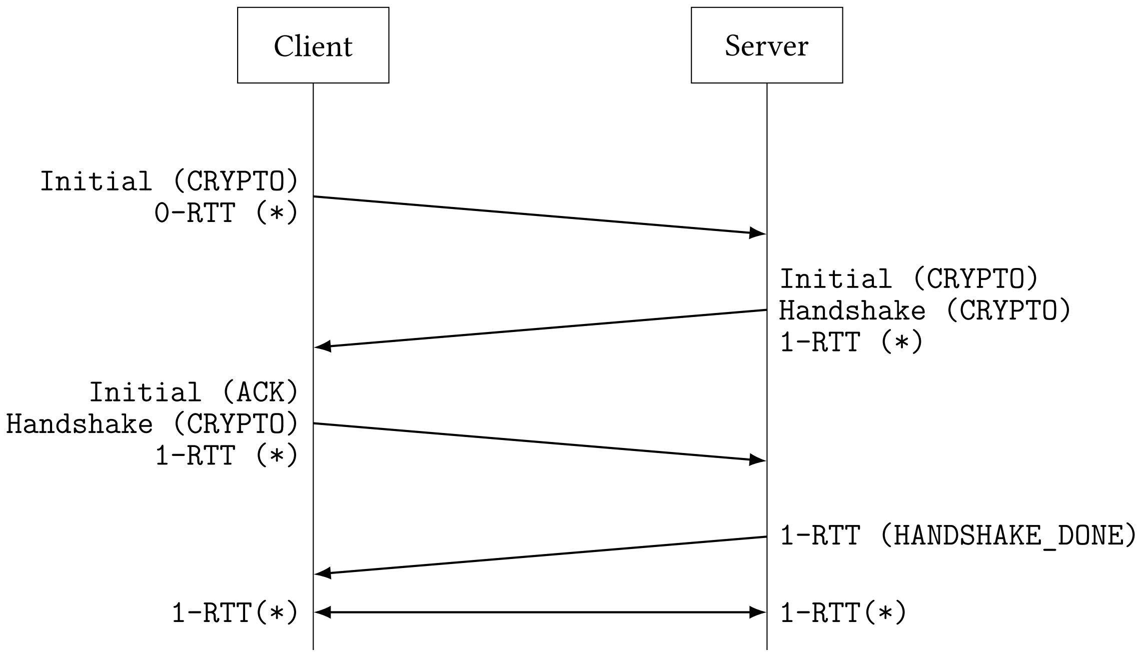

Figure 1.3: QUIC handshake sequence diagram.

The exchanges that form the QUIC handshake are shown in the diagram in figure 1.3.

In the diagram, the names "Initial", "Handshake", "0-RTT" and "1-RTT" refer to packet types, which differ in their header structure and packet numbering. The words in parentheses refer to the frame types contained within the packet. Cryptographic negotiation is done via `CRYPTO` frames. The asterisks indicate packets that may contain application data.

Applications may choose to transmit data during the connection handshake, with some limitations, trading some security guarantees for reduced latency:

- The client may send data to the server in its very first message, using a 0-RTT packet. This mechanism requires previous communication between the two endpoints, since it relies on both client and server remembering protocol parameters negotiated in a previous connection. Connections using 0-RTT may however be vulnerable to replay attacks.
- The server may choose to send application data to the client before it receives the final cryptographic handshake messages that confirm the identity of the client.

After the server sends out the `HANDSHAKE_DONE` frame, data packets can be transmitted freely.

Compared to QUIC's handshake, which lasts at most 2 RTTs thanks to the combination of the transport and TLS exchanges, TCP requires first a three-way handshake, then three more exchanges to complete the TLS handshake, for a total of six exchanges (i.e. three RTTs) before any application data can be sent.

1.4.2 Connection migration

QUIC connections are identified by connection IDs rather than by a four-tuple of two IPs and two ports. The use of IDs allows QUIC to migrate connections from one connection path to another; in other words, one endpoint may change its IP or port without aborting the ongoing connection, for example due to NAT rebinding. Such migration is not possible with TCP connections, as they are statically identified by their four-tuple of IPs and ports.

For the sake of security, each connection is not associated to a single fixed ID, but rather to a set of IDs. Endpoints autonomously associate packets to any one of the connection IDs available to them, preventing a traffic observer from identifying packets that are part of the same connection.

The pool of available connection IDs is updated dynamically. New connection IDs are issued through `NEW_CONNECTION_ID` frames, and old ones can be retired through `RETIRE_CONNECTION_ID` frames.

1.4.3 Operations on connections

The following are some of the main operations on connections operable by client and server. An application acting as a client can:

- Open a connection,
- Enable Early data when available,
- Be informed whether Early data has been accepted or rejected by the server.

An application acting as a server can:

- Listen for incoming connections.

Either role can:

- Configure minimum values for the number of permitted streams of each type,
- Set flow control limits for individual streams and the connection,
- Keep a connection from silently closing by generating `PING` frames.

1.5 Loss detection

Loss detection is described in detail in [RFC9002]

QUIC packet numbers are strictly monotonically increasing, and are decoupled from the data offsets of streams. The receiver determines delivery order (for each stream) based on the offset value within `STREAM` frames.

Whenever an `ACK`-eliciting packet is declared lost, the sender determines what data need to be retransmitted, and places them in new frames, in a new packet (with a higher packet number).

Packet numbers are used within QUIC `ACK`s, which contain many ranges of them – this is unlike TCP, where `ACK`s indicate acknowledged data using sequence numbers (i.e. byte offsets).

Overall, there are several advantages to the design choice of using monotonically-increasing packet numbers:

- It removes an ambiguity present in TCP regarding the acknowledgments of retransmitted packets: in TCP, if a packet has been sent multiple times, it is impossible for the sender to know for certain which one a newly arrived `ACK` refers to - unless packet timestamps are available.
- Since `ACK`s are never ambiguous, they can always be used by a sender to derive estimates of the transmission delay.
- Spurious retransmissions are easily detected by senders: if a packet was assumed to be lost and spuriously retransmitted, the `ACK` of the retransmitted one will contain the information that both that packet and its previous copy were in fact received. This information can be useful for the sender, as it may decide to roll back the congestion control changes made because of that supposed loss.
- The logic of QUIC loss mechanisms is generally simpler to implement.

A QUIC sender determines whether packets are lost based on missing acknowledgments, similarly to TCP. QUIC's acknowledgment-based loss detection explicitly seeks to implement the ideas behind a multitude of TCP mechanisms, such as:

- RTO: In standard TCP, the RTO (retransmission timeout) [RFC6298] refers to the time interval that a sender will wait for the `ACK` of a packet before assuming it was lost and retransmitting it. The wait is dynamically calculated based on the endpoint's RTT estimate and RTT variance. The expiration of the RTO is interpreted as a sign of congestion, and triggers a reset of the congestion window.
- Fast Retransmit [RFC5681] is a standard TCP technique by which a TCP sender will immediately retransmit a packet if it receives three `ACK`s that signal its failed delivery, rather than wait for the RTO to expire. The idea is that a TCP receiver that is missing a packet will keep sending `ACK`s requesting that same packet, so a sender that receives many duplicate `ACK`s can infer that only that one packet was lost, while others are successfully reaching the receiver. Thus, retransmission can occur immediately, and the congestion window is merely halved rather than fully reset to its initial value.
- Early Retransmit [RFC5827] is a mechanism by which the number of duplicate `ACK`s

required to trigger Fast Retransmit is reduced in certain circumstances (e.g. with a small congestion window, or a short transfer).

- TLP is part of the [RFC8985] loss detection algorithm. TLP (Tail Loss Probe) is a mechanism that allows the fast recovery of lost packets at the "tail" of transmissions. Such losses would normally trigger the RTO, as a consequence of being last, seemingly indicating that there is serious congestion - when it may not be necessarily true. TLP sends probe segments right after tail packets, so that the probe segments' `ACK`s may be used to trigger Fast Retransmit for any lost tail packets.

In QUIC, a packet is declared lost if it meets all of the following criteria:

- It is "in flight" (i.e. unacknowledged), and
- It was sent before a packet that has now been acknowledged, and
- The packet violates at least one the of QUIC thresholds:
  - The "time threshold", meaning that the packet was sent too long ago.
  - The "packet threshold", meaning it was sent too many packets before an acknowledged packet. This results in an effect similar to that of TCP's Fast Retransmit mechanism.

Having two thresholds provides tolerance against both delays and reordering. The suggested packet threshold is 3, whereas the time threshold must be calculated based on the RTT estimate and RTT variance.

Additionally, QUIC uses a probe timeout mechanism (PTO) to help connections recover from the loss of tail packets and acknowledgments, taking inspiration from TCP's RTO and TLP mechanisms. Whenever an `ACK`-eliciting packet is transmitted, the sender sets a PTO timer, whose duration is calculated based on the current RTT estimate, RTT variance, and maximum `ACK` delay. This duration represents the amount of time that a sender ought to wait for the `ACK` of a sent packet.

When the PTO expires, the sender sends one or two `ACK`-eliciting "probe datagrams" (the exact number depends on the implementation, though sending multiple packets increases the mechanism's resilience to packet drops). Probe datagrams typically contain new application data, not retransmitted data (unless there is no new data to send), and are sent regardless of the congestion window.

Unlike the RTO, the expiration of the PTO is not used an an indication of packet loss, and does not (normally) trigger a retransmission. The purpose of the mechanism is to make the receiver emit new `ACK`s, to provide the sender with updated information regarding the state of the unacknowledged packets that caused the PTO expiration in the first place.

The PTO uses an "exponential backoff" approach, meaning it doubles every time it expires, in order not to exacerbate network congestion.

A QUIC sender establishes that the network is experiencing "persistent congestion" only when all packets sent over a certain duration of time are deemed lost. This situation is interpreted by QUIC as the equivalent of RTO expiration in TCP.

1.6 Congestion control

Network congestion occurs when the traffic in a network exceeds the network's capacity to dispatch packets. In a congested network, IP packets are delayed and lost in network bottlenecks due to buffer overflows at intermediate routers, causing overall performance degradation.

To prevent congestion, Transport protocols such as TCP and QUIC make use of congestion control algorithms, which determine the amount of data that can be injected into a network based on a variety of factors. A congestion algorithm determines a congestion window (cwnd), which acts as a limit to the amount of data "in flight", i.e. unacknowledged, preventing packets from being indiscriminately injected into the network. Different algorithms vary in the ways they calculate the cwnd value, and in what events they react to to update it.

Congestion control is a concept distinct from flow control. The former consists in endpoint strategies to reduce the risk of overwhelming the network with packets, whereas the latter consists in the mechanisms by which sender and receiver operate in order not to overwhelm the receiver's buffer capacity. In flow control, the receiving endpoint advertises a "receiver window" (or rwnd), which limits the sender's output. Together, congestion window and receiver window result in an effective window of min(cwnd, rwnd).

The QUIC specification suggests a congestion controller similar to TCP NewReno, though endpoints may opt to use a different algorithm. As the object of this thesis is the implementation of a Hybla congestion controller for QUIC, and the algorithms of Hybla and NewReno have similar structure, the RFC's considerations regarding congestion control are worth examining.

According to the QUIC specification, a NewReno-like congestion controller for QUIC should transition between the following states:

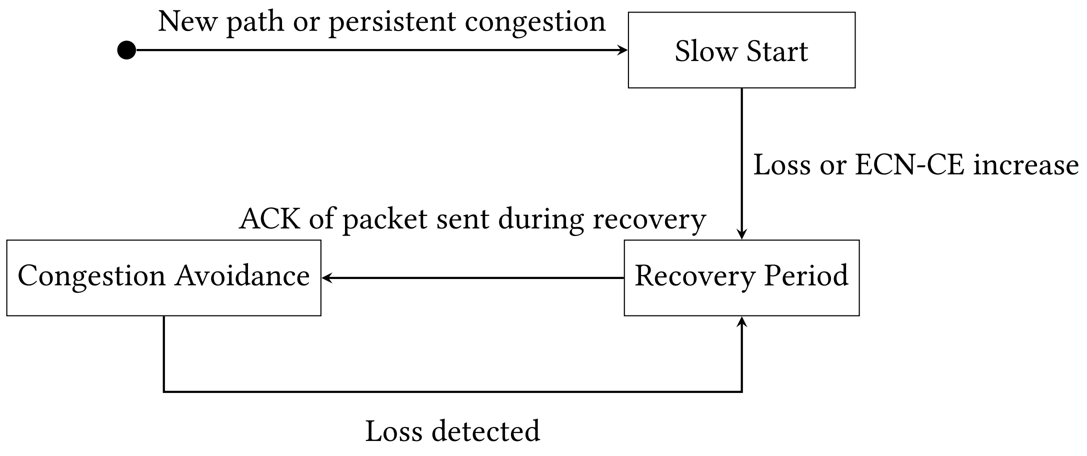

Figure 1.4: States of a NewReno-like QUIC congestion controller as per RFC 9002.

The sender begins in the slow start phase: with each acknowledgment the congestion window increases by the number of bytes acknowledged. Macroscopically, this results in an exponential growth over time of the cwnd. If a loss is detected, or an explicit congestion notification (ECN, [RFC3168]) is reported by one of the intermediate nodes of the network, the sender goes into a state of recovery.

In the recovery state, the sender halves the congestion window, and sets the slow start threshold to that same value. Implementations may choose to protract in time the recovery phase, making the cwnd reduction occur gradually, or they may apply it immediately. Recovery ends when a packet sent during recovery is acknowledged, at which point the sender enters the congestion avoidance phase.

In congestion avoidance, the congestion window still increases with each acknowledgment, but in a more limited way. Specifically, it should increase with an approach that results in a cwnd increase of, at most, the maximum size of one datagram every time that an amount of bytes equivalent to the congestion window is acknowledged.

When the network is considered to be in “persistent congestion”, the sender’s cwnd is reset to its minimum value, and the sender returns to the slow start phase, analogously to when a TCP sender experiences a RTO expiration.

1.7 Packet pacing

The QUIC specification states that senders should pace the sending of packets based on input from the congestion controller. This approach is intended to prevent short-term congestion and losses caused by sending many packets into the network without any delay between them (”bursts”). Pacing does not apply to packets containing only `ACK` frames, as timely delivery of `ACK` frames is essential for connection performance, and `ACK`s frames are also so small that they cannot form significant bursts.

A perfectly-paced sender spreads packets evenly over time. A pacing rate that achieves this can be computed by dividing the congestion window by the RTT.

Chapter 2 TCP Hybla

TCP congestion control is one of the most debated topics in networking, with many proposals presented over the years in the literature. The algorithms actually implemented are relatively few, with about ten being currently available in Linux, and the default being Cubic.

Generally speaking, these algorithms can be divided in two categories, depending on the parameter used as a congestion indicator:

- Losses: this is the case of NewReno [RFC6582], Cubic [RFC8312], and many other important congestion control algorithms. When packet loss is detected, the congestion window is reduced to mitigate congestion.
- Delays: algorithms in this category are Fast [FAST] and BBR [BBR]. Rather than waiting for packet loss to occur, these algorithms use variations in the RTT estimate as a measure of network congestion.
  The idea is that, as a network approaches congestion, queuing delays increase; this acts as an early warning signal, indicating that the congestion window should be adjusted. BBR (Bottleneck Bandwidth and Round-trip propagation time), developed by Google, expands on this idea by not only observing delays, but also loss rate and rate of successful delivery, and using this information to build a model of the network. From its model, it can derive an estimate of the maximum available bandwidth, and adjust its sending rate accordingly.

All congestion control algorithms periodically update the value of the congestion window (cwnd), which is used by the protocol to limit the number of in-flight packets, to prevent network congestion.

Hybla is a NewReno variant designed to cope with long round-trip times (RTTs), such as those in connections with GEO satellite links, thus solving NewReno's "RTT-unfairness" problem. Hybla was originally proposed in a 2004 paper by Carlo Caini and Rosario Firrincieli [Caini_2004] and is currently among the few variants available in the Linux official kernel (the default value of Cubic can be overridden by users by means of `sysctl` commands).

NewReno's RTT-unfairness problem Internet protocols such as TCP were designed for terrestrial networks, where the end-to-end delay between source and destination was limited to a few tens of milliseconds. Even with distant locations, e.g. Bologna and Los Angeles, delays rarely go above 200 ms.

However, if the path between source and destination includes a radio link, the RTT increases considerably. For example, in the presence of GEO satellite links, the RTT will increase to 500 ms at minimum, and typically go even beyond 600 ms.

NewReno, designed for terrestrial usage, struggles with such high RTT values — its congestion window updates are based on the arrival of `ACK`s; longer RTTs cause `ACK`s to be less frequent, which in turn makes the congestion window grows more slowly.

Using standard TCP versions, connections with high RTTs are heavily penalized in comparison to fully wired ones. In scenarios with competing connections, it is said that high-RTT connections "starve", as the congestion windows of short-RTT connections grow quickly and allow them to make use of the majority of the available bandwidth.

2.1 Window updating

2.1.1 Update rules of typical TCP algorithms

The congestion window updating of TCP congestion algorithms such as Tahoe, Reno, and NewReno follows a behavior that is split between a "slow start" phase and a "congestion avoidance" phase.

- In the slow start phase, the connection starts off with a small congestion window (either because the connection has just begun, or because of a loss). With each acknowledgment, the congestion window grows by an amount equal to the data acknowledged, which macroscopically results in an exponential growth over time. This continues until either there is a loss or the window reaches the slow start threshold (ssthresh), at which point the protocol shifts to the congestion avoidance phase. The value of the slow start threshold is initially set to a large value, then, whenever a packet is lost, it is updated to one half of the current cwnd value.

- In the congestion avoidance phase, the growth of the congestion window is linear with time. The change from exponential growth to linear growth prevents the network from becoming abruptly overloaded, while maintaining a steady increase in the transmission rate of the connection, allowing the sender to probe the network for possible additional bandwidth.

The exponential/linear growth of the cwnd is the macroscopic result of individual changes occurring at the reception of each `ACK`: in the aforementioned congestion control algorithms, the following rules are applied upon receipt of each acknowledgment:

$$W_{i+1} = \begin{cases} W_i + 1 & \text{SS} \\ W_i + 1/W_i & \text{CA} \end{cases}$$

Where W_i is the new congestion window value upon reception of the i-th `ACK`. Values are given in MSS units, which correspond to the maximum amount of bytes that can be put into a TCP segment. The rules can be explained as follows:

- During the SS phase, the congestion window is increased by MSS bytes per every `ACK` received.
- During the CA phase, when a number of `ACK`s cumulatively acknowledge an amount of bytes equal to the size of the congestion window, the window is increased by approximately MSS bytes.

These rules can also be expressed using a continuous model; the congestion window over time $W(t)$ can be defined as:

$$W(t) = \begin{cases} 2^{t/\text{RTT}} & 0 \leq t < t_{\text{ssthresh}} \\ \dfrac{t - t_{\text{ssthresh}}}{RTT} + \text{ssthresh} & t \geq t_{\text{ssthresh}} \end{cases}$$

Where t_{ssthresh} is the time at which the ssthresh is reached, equal to $\text{RTT} \cdot \log_2 \text{ssthresh}$.

In both cases, the RTT appears as a denominator, which makes evident that higher RTTs result in a slower cwnd increase rate.

2.1.2 Update rules of the TCP Hybla algorithm

The throughput of a connection, i.e. the amount of segments transmitted per second, can be expressed as:

$$B(t) = W(t)/\text{RTT}$$

(For convenience, this assumes that the effective window is equal to the congestion window, meaning the sender's output is limited by congestion control rather than flow control).

Hybla seeks to make transmission performance independent of RTT. This can achieved with two modifications [Caini_2004]:

- A time scale modification: if the RTT is larger, the cwnd value must increase faster, to compensate for the fact that larger RTTs make cwnd updates necessarily less frequent.
- A multiplication by RTT, to compensate for the division by RTT in the throughput's expression.

More specifically, Hybla works by increasing the performance of a long-RTT connection in such a way that it matches that of a given reference connection (e.g. a wired connection) of round trip time RTT_0.

Hybla introduces a quantity called **normalized round trip time**, ρ, which helps in applying the two modifications. It is defined as:

$$\rho = \frac{\text{RTT}}{\text{RTT}_0}$$

Where RTT_0 is the RTT of the reference connection whose performance Hybla aims to match.

The aforementioned two modifications can be achieved through the following changes to the continuous model of the congestion window:

- Multiplying the time t by ρ.
- Multiplying the entire expression by ρ.

The result is as follows:

$$W^{H(t)} = \begin{cases} \rho \cdot 2^{\rho t/\mathrm{RTT}} & 0 \leq t \leq t_{\rho \cdot \mathrm{ssthresh}} \\ \rho \left(\rho \dfrac{t - t_{\rho \cdot \mathrm{ssthresh}}}{\mathrm{RTT}} + \mathrm{ssthresh} \right) & t \geq t_{\rho \cdot \mathrm{ssthresh}} \end{cases}$$

Where $t_{\rho \cdot \mathrm{ssthresh}}$ is the time at which the congestion window reaches the value of $\rho \cdot \mathrm{ssthresh}$. This value is the same regardless of RTT, being equal to $\mathrm{RTT}_0 \cdot \log_2 \mathrm{ssthresh}$.

These equations form the continuous model of the Hybla congestion window over time. Using this model, the throughput results in:

$$B^{H(t)} = \frac{W^{H(t)}}{\mathrm{RTT}} = \begin{cases} \dfrac{2^{t/\mathrm{RTT}_0}}{\mathrm{RTT}_0} & 0 \leq t < t_{\rho \cdot \mathrm{ssthresh}} \\ \dfrac{1}{\mathrm{RTT}_0} \left(\dfrac{t - t_{\rho \cdot \mathrm{ssthresh}}}{\mathrm{RTT}_0} + \gamma \right) \end{cases}$$

It can be observed that this value is independent of RTT, and matches the transmission rate of the reference connection. This means that, no matter what the true RTT of the connection is, Hybla will guarantee the same transmission rate as that of a NewReno connection with round trip time equal to RTT_0.

The continuous model for the Hybla congestion window can be converted into the following discrete rules, to be applied whenever an ACK is received:

$$W_{i+1}^{H} = \begin{cases} W_i^H + 2^\rho - 1 & SS \\ W_i^H + \rho^2 / W_i^H & CA \end{cases}$$

It is worth noting that Hybla requires the initial cwnd, ssthresh, and transmission buffer size values to be multiplied by ρ. Additionally, when implementing the algorithm, it is appropriate to set the minimum value of ρ to 1.0, so that the modifications are not applied if $\mathrm{RTT} < \mathrm{RTT}_0$.

QUIC congestion algorithms rely on a congestion window analogous to that of TCP, which means that the presented reasoning, as well as the resulting rules for congestion window updates, can be applied to QUIC congestion control.

2.2 Loss recovery mechanisms and packet pacing

TCP Hybla does not consist only of its congestion control rules. In addition to the algorithm discussed, Hybla envisages the following features:

- Use of the SACK option: TCP Hybla suggests the use of this option, as it allows the recovery of multiple lost packets within a single RTT. This is especially useful in TCP Hybla, as the higher cwnd makes it more likely to have many losses within the same window. Linux uses SACK by default.
- Use of packet timestamps: in TCP, a loss is declared in the event that there is no feedback from the receiver. The wait before such a loss is declared is referred to as RTO (retransmission time out), and it is computed based the sender's estimate of the connection's RTT, which in turn is derived from the arrival time of `ACK`s. Generally, if the retransmission timer hits the RTO, retransmission occurs, the RTO is doubled, and the timer restarts; this cycle continues until data delivery is confirmed. The resulting RTO will eventually be recomputed and updated based on the arrival time of new ACKs. However, if packets are not timestamped, this computation can only be made with an `ACK` of a non-retransmitted packet, thus postponing the RTO update, which can severely penalize long-RTT connections and potentially lead to long stalls. On the other hand, with timestamped packets, any `ACK` can give reliable arrival-time information to recompute the RTO, which makes the use of timestamps a preferable choice in TCP Hybla. Moreover, timestamps allow a receiver to distinguish between original and retransmitted data. The use of timestamps is the default in Linux.
- Packet pacing: packets within the congestion window should be sent gradually, evenly spread over each RTT, rather than in large bursts that may overwhelm the network's capacity. Hybla's performance boost to long-RTT connections results in larger cwnds, making transmission "burstiness" more likely to result in network congestion. This point, originally proposed in Hybla in 2004, was initially refused by Linux maintainers, resulting in an implementation of TCP Hybla that lacked one of its components. With pacing as a recommended feature in QUIC, it is evident that there is now a better awareness of the potential issues caused by bursts of packets.

In the context of a QUIC implementation of Hybla, these points are addressed as follows:

- QUIC does not need an analogue of the SACK option, as QUIC acknowledgments are already designed after the ideas of SACK, and by default can selectively acknowledge many intervals of packets.
- QUIC packets are referred using monotonically increasing packet numbers. This means that all packets provide accurate arrival-time information for RTT estimation, in contrast to TCP, where this is only true for the `ACK`s of non-retransmitted packets.
- As previously said, similarly to the original Hybla proposal, the QUIC specification suggests that packets be spaced out evenly rather than sent in bursts.

2.3 Fairness and friendliness

In the context of evaluating the performance of congestion control algorithms:

- *Fairness* is a measure of an algorithm's ability to lead to a fair bandwidth subdivision in the presence of competing connections that use the same congestion control algorithm.
- *Friendliness* measures the same ability, but in the presence of competing connections using other congestion control algorithms.

NewReno and other similar algorithms are "fair" provided that all connections have similar RTTs. Otherwise, when short-RTT and long-RTT connections compete, the former tend to acquire a greater amount of the available bandwidth than the latter. When the RTT disparity is huge, as when a GEO satellite connection has to compete with a terrestrial connection, the long-RTT one "starves" as nearly all the available bandwidth is taken by the short-RTT ones – this is referred to as the "RTT-unfairness problem".

Hybla's aim is to solve this problem, by boosting long-RTT connections to have the same performance as a short-RTT NewReno connection considered as a "reference". This way, fairness among competing connections can be guaranteed as long as the chosen RTT_0 value is comparable to the RTTs of other connections present.

2.4 Choice of RTT_0

The choice of the right RTT_0 value is paramount in Hybla. As the goal of Hybla is to reduce the performance penalization of high-RTT connections, the ideal RTT_0 value in a satellite network would be that of the connection itself, without the additional delay caused by the satellite links.

The ideal value may be hard to estimate, but a conservative choice can still assure a significant performance boost without causing inconvenience to the network. In the literature [Caini_2004], as well as in the Linux implementation of TCP Hybla [Hybla_Linux], the default value is 25ms, which is a reasonable average for terrestrial connections.

Chapter 3 Picoquic

Picoquic is a minimalist implementation of the QUIC protocol, developed in C and created for testing and experimentation purposes by Christian Huitema [Picoquic]. Picoquic development began in June 2017, and since then the project has been constantly updated. The current version implements the specifications given in RFCs 9000, 9001, 9002, and 8999, as well as a number of extensions.

3.1 QUIC API

3.1.1 Sending data

Picoquic provides applications with two ways to send stream data: a “queuing” API and a “just in time” API.

Sending data with the queuing API

Applications can add data to a specific stream by calling the following functions.

```
int picoquic_add_to_stream(picoquic_cnx_t* cnx, uint64_t stream_id, const
↪  uint8_t* data, size_t length, int set_fin);

int picoquic_add_to_stream_with_ctx(picoquic_cnx_t* cnx, uint64_t stream_id,
↪  const uint8_t* data, size_t length, int set_fin, void* app_stream_ctx);
```

Both functions require the stream ID of the desired stream, a pointer to the data to be sent, the length of said data, and a boolean parameter `set_fin` to state whether the data in question mark the end of the stream. The second function also requires the `app_stream_ctx` parameter, which is used in the receive callbacks.

When an application calls the functions above, the provided data are copied to the stream’s internal queue – each stream has its own. The data will then be sent into the network as soon as it is allowed by congestion control, flow control, and stream priority constraints.

Sending data with the just-in-time API

With the just in time API, the application is notified when a stream is available for sending. To accomplish this, the application first marks the desired stream as "active", using `picoquic_mark_active_stream`.

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

Then, when the activated stream is ready, the application will receive the callback `picohttp_callback_provide_data`. Within the callback, the application will then reserve a buffer using `picoquic_provide_stream_data_buffer`, and copy the application data in it.

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

Compared to the queuing API, the just-in-time API does not imply any buffering of the data to be sent, as Picoquic notifies the application at the moment when data can be sent immediately. This approach may not be suitable to applications where the data to be sent is generated asynchronously, as such applications will be forced to buffer the data themselves while waiting for the Picoquic callback. On the other hand, applications that are able to generate data upon receiving the callback (e.g. by reading it from a file) will benefit from the just-in-time approach.

3.1.2 Receiving data

In QUIC, data transferred on a stream are to be delivered in order, so it is necessary to buffer incoming data until they can be delivered in the correct order to the application. Once ordered delivery can be guaranteed, the application will receive a `picoquic_callback_stream_data` callback, which will provide the received data.

3.1.3 Closing streams

When a sender intends to end transmission on a stream, it sets to 1 a specific parameter in its last call to either `picoquic_add_to_stream_with_ctx` (if using the queuing API) or `picoquic_provide_stream_data_buffer` (if using the just-in-time API), thus causing the FIN bit to be set to 1 on the final frame.

A sender may also abruptly close a stream by resetting it, using the function:

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

The reset will immediately stop all transmission on the stream, be it new or retransmitted data.

3.1.4 Unreliable datagram extension

Picoquic also supports the unreliable datagram extension described in [RFC9221]. This extension allows endpoints to exchange QUIC datagrams that, contrary to the reliable data flow of streams, are not reliable, and thus are more suitable to applications that transmit real-time data.

Like stream data, QUIC datagrams can be sent either with a queuing API or a just-in-time API. To queue a datagram:

```
int picoquic_queue_datagram_frame(picoquic_cnx_t* cnx, size_t length, const
↪   uint8_t* bytes);
```

To send datagrams just-in-time, the application first indicates its readiness to send datagrams using `picoquic_mark_datagram_ready`. This enables it to later receive the callback `picoquic_callback_prepare_datagram`, in which the application may then reserve a buffer using `picoquic_provide_datagram_buffer_ex`, and place the datagram's bytes inside this buffer.

On the receiving side, an application will receive the `picoquic_callback_datagram` callback whenever a datagram is received.

3.2 Provided software

The Picoquic project includes a number of auxiliary programs. Relevant to this thesis are `picoquicdemo` and `picoquic_sample`, which were relevant in the testing and evaluation of the work that is the object of this thesis.

picoquicdemo

The command to use `picoquicdemo` is as follows, in both server and client endpoints:

```
picoquicdemo <options> [server_name [port [scenario]]]
```

The program automatically determines whether it must run as a server or as a client based on the provided arguments. To create a client endpoint, one must provide server name and port. To create a server endpoint, it is enough to provide only the desired port.

Through the `-a` option, the user can select the application-level protocol that `picoquicdemo` will use in the performance evaluation of the QUIC connection between server and client.

One such protocol is quicperf, which is currently defined as an Internet Draft [quicperf]. Another is picoquicperf, a fork of quicperf implemented by fellow student Luca Andreetti as part of his work [Andreetti_2025], which adds the option to send an as much data as possible for a given time duration. The latter was extensively used in the debugging and validation of the work of this thesis.

Each of these protocols has its own set of supported test characteristics; users can provide their desired test parameters through the `scenario` string, which has different formats and fields depending on the chosen protocol. For picoquicperf, the scenario string is composed of one or more concatenated substrings, each one with the following structure:

```
<cnx_type>:*<count>:<stream_id>:<prev_stream_id>:<cnx_details>;
```

Each substring describes the behavior of a stream. For example, inputting the string `T:*1:0:-:10s; D:*1:2:-:60000:60000` means that, on execution, as much data as possible will be sent to the peer for 10 seconds on a stream of ID 0, and, concurrently, on a bidirectional stream of ID 2, 60000 bytes will be transmitted from client to server and viceversa.

A full description of the meaning of each field of the picoquicperf `scenario` string can be found in Luca Andreetti's thesis.

picoquic_sample

`picoquic_sample` is a simple file transfer program. To create a server endpoint:

```
picoquic_sample server [port] [cert] [key] [folder]
```

Where:

- `port` is the desired port,
- `cert` is the certificate file, to verify the authenticity of the server's identity.
- `key` is a file with the server's private key. Together with cert, these two parameters are used to implement TLS.
- `folder` is the folder containing the files that a client may request.

To create a `picoquic_sample` client endpoint:

```
picoquic_sample client [server] [port] [dest_folder] [queried_file]
```

Where:

- `server` is the IP of the server endpoint.
- `port` is the port on which the server listens.
- `dest_folder` is the path of the folder where received files will be placed.
- `queried_file` is the name of the requested file.

3.2.1 Additional options

Either program, on top of the parameters and options specific to it, can receive additional options, which are forwarded to the underlying Picoquic library. Some of these options are:

- `-G`: to specify the desired congestion control algorithm. This is done by providing a string identifier, e.g. "`reno`" (which corresponds to Picoquic's NewReno implementation), "`bbr`", "`hybla`".
- `-x`: to specify the maximum number of connections.
- `-q`: to provide a folder for packet logging, in the form of .qlog files.
- `-W`: to specify the maximum congestion window size.

Chapter 4 Implementation of QUIC Hybla

The current implementation of Picoquic includes a number of congestion control algorithms. They are implemented with a modular approach, each one having their specific logic and definitions all contained within a dedicated source file.

The work of implementing the Hybla congestion control strategy and incorporating it into the existing Picoquic codebase consisted in:

- Creating `hybla.c`, containing all the logic and definitions specific to the Hybla algorithm.
- Making Hybla-specific additions to select locations of the codebase, so that Hybla can be selected as one of the congestion control algorithms available, and configured as needed.
  - Hybla can receive the value of RTT_0 as an external parameter, so that it can be changed without the need to recompile the code. By default, this value is set to 25ms, as per the Linux implementation of TCP Hybla.
  - Hybla can also receive an `INITIAL_SSTHRESH` parameter, to set the starting value of the slow start threshold. Typically, congestion control algorithms in Picoquic have their ssthresh value initially set to `UINT64_MAX`, meaning that the initial slow start phase continues until there is a loss. However, it was observed in preliminary tests that, if ρ is really high, the Hybla cwnd may jump to extremely high values during slow start. Setting `INITIAL_SSTHRESH` helps ensure that, even in such scenarios, the cwnd never reaches an unreasonably tall peak.

Studying the Linux implementation of TCP Hybla was particularly helpful in the implementation of Hybla in Picoquic; its source code, by Daniele Lacamera, can be found at [Hybla_Linux].

The implemented Hybla algorithm has been presented to the Picoquic maintainers, and as of this writing is in the process of being fully integrated in the official [Picoquic] repository.

4.1 Congestion control algorithms in Picoquic

QUIC specifications do not mandate the use of a specific congestion control algorithm, provided that the one adopted is compatible with a standard QUIC receiver. This approach follows the one of Linux in regard to TCP, where, starting from Kernel 2.6.0, the core code of the protocol is decoupled from the congestion control algorithm, letting users experiment with new possibilities and use their preferred solution.

To achieve this aim, all congestion control algorithm implementations in Picoquic are represented by a struct variable of type `picoquic_congestion_algorithm_t`, containing the following fields:

- A string ID,
- An integer ID,
- Four function pointers, to functions that determine the behavior of the congestion control algorithm. These functions are referred to as “init”, “notify”, “delete” and “observe” respectively.

For example, in the case of the Cubic congestion control algorithm, the struct is:

```
#define picoquic_cubic_ID "cubic"
picoquic_congestion_algorithm_t picoquic_cubic_algorithm_struct = {
    picoquic_cubic_ID,
    PICOQUIC_CC_ALGO_NUMBER_CUBIC,
    cubic_init,
    cubic_notify,
    cubic_delete,
    cubic_observe
};
```

Where `cubic_init`, `cubic_notify`, `cubic_delete`, and `cubic_observe` are function pointers. This struct is always defined as `extern`, which means it has global scope.

When an endpoint is initialized, and the desired congestion control is chosen, the `picoquic_congestion_algorithm_t` variable of the selected congestion control is saved within the endpoint’s QUIC context. From that point onward, other functions will be able to call the functions of the chosen congestion control through the pointers inside the struct.

The fact that all congestion control implementations encapsulate their functionality within a struct of the same type allows internal Picoquic functions to use the functionality of the chosen congestion control in a transparent way, meaning that they do not need to know which specific algorithm has been selected, which gives an great advantage in terms of code modularity and reliability.

4.2 Implementation of hybla.c

The implementation of Hybla.c consists of the four functions mentioned previously, as well as a number of utility functions.

4.2.1 The Init function and the Hybla state structure

```
static void picoquic_hybla_init(picoquic_cnx_t * cnx,
        picoquic_path_t* path_x, uint64_t current_time) {

    /* Initialize the state of the congestion control algorithm */
    picoquic_hybla_state_t* hybla_state = (picoquic_hybla_state_t*)
            malloc(sizeof(picoquic_hybla_state_t));

    if (hybla_state != NULL) {
        picoquic_hybla_reset(hybla_state, path_x);
        path_x->congestion_alg_state = hybla_state;
    }
    else {
        path_x->congestion_alg_state = NULL;
    }
}
```

This function is called upon the initialization of the sender endpoint, if Hybla has been selected. Its purpose is to initialize `hybla_state`, a struct that holds the state of the Hybla algorithm.

```
typedef struct st_picoquic_hybla_state_t {
    picoquic_hybla_alg_state_t alg_state;
    uint64_t cwin;
    uint64_t ssthresh;
    uint64_t recovery_start;
    uint64_t recovery_sequence;

    int rtt0;
    double rho;
    int rho_is_initialized;
    uint64_t rtt_used_for_rho;

    double increment_frac_sum;
} picoquic_hybla_state_t;
```

- `alg_state` holds the current state of the algorithm, which can be either `picoquic_hybla_alg_slow_start` or `picoquic_hybla_alg_congestion_avoidance`, analogously to Picoquic's NewReno implementation.
- `cwin` and `ssthresh` hold the current congestion window and slow-start threshold values, respectively.
- `recovery_start` holds the time recovery was last entered, and `recovery_sequence` holds the number of the last sent packet at the time of `recovery_start`. These values are referenced when specific events cause a call to the notify function, to determine whether recovery should be triggered.
- `rtt0` holds the value of Hybla's RTT_0 parameter.
- `increment_frac_sum` is used in the congestion window updating algorithm to accumulate the decimal parts resulting from the calculation of each new congestion window value. Whenever this value becomes greater or equal than one, a unit is subtracted from it and added to the cwnd.

4.2.2 Observe and delete functions

Hybla's observe function provides the current state of the congestion control algorithm and the current ssthresh value.

```
void picoquic_hybla_observe(picoquic_path_t* path_x, uint64_t* cc_state,
↪   uint64_t* cc_param) {

    picoquic_hybla_state_t* hybla_state =
    (picoquic_hybla_state_t*)path_x->congestion_alg_state;

    *cc_state = (uint64_t)hybla_state->alg_state;
    *cc_param = (hybla_state->ssthresh == UINT64_MAX) ? 0 :
    ↪   hybla_state->ssthresh;
}
```

The delete function is called at the termination of the sender to release the state of the congestion control algorithm.

```
static void picoquic_hybla_delete(picoquic_path_t* path_x) {

    picoquic_hybla_state_t* hybla_state = path_x->congestion_alg_state;

    if (path_x->congestion_alg_state != NULL) {
        //...
        free(path_x->congestion_alg_state);
        path_x->congestion_alg_state = NULL;
    }
}
```

4.2.3 The notify function

The notify function is the most complex, as it contains the core of the congestion control logic. The function is called whenever specific events happen, and receives in input a notification parameter that indicates the nature of the event – for example, a notification of value `picoquic_congestion_notification_acknowledgement` indicates that an `ACK` has been received. Whenever the event is associated to the reception of an acknowledgment, the function receives the information associated to the `ACK` in question through the parameter `ack_state`.

Inside the function, a switch selects the actions to be taken on the basis of the notification received.

```
static void picoquic_hybla_notify(
picoquic_cnx_t * cnx,
picoquic_path_t* path_x,
picoquic_congestion_notification_t notification,
picoquic_per_ack_state_t * ack_state,
uint64_t current_time)
{
    picoquic_hybla_state_t* hybla_state =
     ↪  (picoquic_hybla_state_t*)path_x->congestion_alg_state;

    path_x->is_cc_data_updated = 1;

    if (hybla_state != NULL) {
        switch (notification) {
            case picoquic_congestion_notification_acknowledgement:
            // ...
            case picoquic_congestion_notification_seed_cwin:
            // ...
            case picoquic_congestion_notification_ecn_ec:
            case picoquic_congestion_notification_repeat:
            case picoquic_congestion_notification_timeout:
            // ...
            case picoquic_congestion_notification_spurious_repeat:
            // ...
            case picoquic_congestion_notification_rtt_measurement:
            // ...
            case picoquic_congestion_notification_cwin_blocked:
            // ...
            case picoquic_congestion_notification_reset:
            // ...
            default:
            break;
        }

        // Update packet pacing
        // ...
    }
}
```

The case picoquic_congestion_notification_acknowledgement is ran whenever an acknowledgment is received. This is where the congestion window value is regularly updated, according to the algorithm's rules.

```
case picoquic_congestion_notification_acknowledgement:

// First, set the congestion window's minimum value to half of the max.
↪  possible window, calculated based on RTT and bandwidth estimates.

if (hybla_state->alg_state == picoquic_hybla_alg_slow_start &&
!path_x->is_ssthresh_initialized) {

    uint64_t max_win = path_x->peak_bandwidth_estimate * path_x->smoothed_rtt
    ↪  / 1000000;
    uint64_t min_win = max_win / 2;

    if (hybla_state->cwin < min_win) {
        hybla_state->cwin = min_win;
        path_x->cwin = hybla_state->cwin;
    }
}

if (path_x->last_time_acked_data_frame_sent >
↪  path_x->last_sender_limited_time) {
    // Apply different rules based on the current congestion control phase
    ↪  (SS or CA)
    switch (hybla_state->alg_state) {
        case picoquic_hybla_alg_slow_start:

        // Apply slow start rules
        // ...

        break;
        case picoquic_hybla_alg_congestion_avoidance:
        default: {
            // Apply congestion avoidance rules
            // ...

            break;
        }
    }

    // Set the updated congestion window into the current path's context, so
    ↪  that it may be used by picoquic to limit sending.
    path_x->cwin = hybla_state->cwin;
}

break;
```

Slow start

Hybla's slow start phase is implemented by the following block:

```
case picoquic_hybla_alg_slow_start:

update_rho(hybla_state, path_x);

double increment_in_mss = pow(2.0, hybla_state->rho) - 1.0;
double increment = ack_state->nb_bytes_acknowledged * increment_in_mss;
uint64_t increment_int_part = floor(increment);
double increment_frac_part = increment - increment_int_part;

uint64_t total_increment = increment_int_part;

hybla_state->increment_frac_sum += increment_frac_part
if (hybla_state->increment_frac_sum >= 1.0) {
    total_increment += 1;
    hybla_state->increment_frac_sum -= 1.0;
}

// If the SS increment would make cwin exceed ssthresh, process the first
↪  chunk of bytes according to SS, and the remaining ones according to CA:
```

```
if (hybla_state->cwin + total_increment > hybla_state->ssthresh) {

    uint64_t excess_increment =
        hybla_state->cwin + total_increment - hybla_state->ssthresh;
    uint64_t excess_bytes =
        floor(ack_state->nb_bytes_acknowledged * excess_increment /
         ↪  total_increment);

    // Handle ssthresh bytes according to SS
    hybla_state->cwin = hybla_state->ssthresh;

    // Handle remaining bytes according to CA
    double ca_increment_from_excess =
    picoquic_hybla_get_raw_ca_increment(hybla_state, path_x, excess_bytes);

    uint64_t ca_increment_int_part = floor(ca_increment_from_excess);
    double ca_increment_frac_part = ca_increment_from_excess -
     ↪  ca_increment_int_part;

    hybla_state->cwin += ca_increment_int_part;

    hybla_state->increment_frac_sum += ca_increment_frac_part;

    if (hybla_state->increment_frac_sum >= 1.0) {
        hybla_state->increment_frac_sum -= 1.0;
        hybla_state->cwin += 1;
    }
}
// Else, handle all bytes as per SS
else
hybla_state->cwin += total_increment;

// If cwin exceeds ssthresh, go to CA
if (hybla_state->cwin >= hybla_state->ssthresh) {
    hybla_state->alg_state = picoquic_hybla_alg_congestion_avoidance;
}
break;
```

At the very start of the slow start section, the function update_rho is called to recalculate the value of ρ as defined in the original proposal:

```
void update_rho(picoquic_hybla_state_t* hybla_state, picoquic_path_t* path_x)
 ↪  {
    if (!path_x->rtt_is_initialized)
    return;

    double new_rho =
    (double) path_x->smoothed_rtt / (__picoquic_hybla_rtt0_param * 1000);
    if (new_rho < 1.0)
    new_rho = 1.0;

    if (!hybla_state->rho_is_initialized || new_rho < hybla_state->rho) {

        // ...

        hybla_state->rho = new_rho;
        hybla_state->rho_is_initialized = 1;
        hybla_state->rtt_used_for_rho = path_x->smoothed_rtt;
    }
}
```

The value of ρ is calculated as the ratio between the current estimated RTT (a smoothed average of the latest measurements, provided by Picoquic) and the externally-provided RTT_0 parameter. The minimum value of ρ is set to 1.0, as a safety measure to prevent Hybla from purposefully slowing down in case that $\text{RTT} < \text{RTT}_0$.

The current value of ρ is overwritten only if the newly calculated value is inferior. A lower value of ρ corresponds to a lower RTT estimate, which is bound to be more representative of the nature of the connection's path. The intent of Hybla is to compensate for the propagation delay only, not queuing delays – in other words, the calculation of ρ uses the minimum estimated RTT since it is the most accurate estimate of the propagation delay.

After calling `update_rho`, the slow start phase calculates the increment to be applied to the congestion window.

Picoquic handles its congestion window in bytes. In the Picoquic implementation of NewReno's slow start, each acknowledgment results in a window increase that is equal to the amount of bytes acknowledged:

```
hybla_state->cwin += ack_state->nb_bytes_acknowledged;
```

The Hybla proposal describes congestion window dynamics using MSS units: where the NewReno SS cwnd increase is stated to be 1, Hybla's is $2^\rho - 1$. So, in Picoquic, Hybla's congestion window increase is that of NewReno, but multiplied by a factor or $2^\rho - 1$:

```
double increment_in_mss = pow(2.0, hybla_state->rho) - 1.0;
double increment = ack_state->nb_bytes_acknowledged * increment_in_mss;
```

```
uint64_t increment_int_part = floor(increment);

uint64_t total_increment = increment_int_part;
```

Since ρ can be fractional, the increment to the congestion window can also be fractional; for this reason, only its integer part is added the `total_increment` variable. The fractional part is instead added to the `increment_frac_sum` variable within Hybla's state. Whenever this variable grows larger than 1, that accumulated unit is moved to `total_increment`.

```
hybla_state->increment_frac_sum += increment_frac_part
if (hybla_state->increment_frac_sum >= 1.0) {
    total_increment += 1;
    hybla_state->increment_frac_sum -= 1.0;
}
```

At this point, an implementation may opt to always add the `total_increment` variable to the congestion window, effectively processing all of the newly acknowledged bytes according to slow start rules. Hybla does not always do this, in order to prevent excessive cwnd increases during slow start; instead, a check is carried out: if the slow start increment is such that it would bring the congestion window above the slow start threshold, then the bytes of the acknowledgment ought to be processed in part according to slow start rules and in part according to congestion avoidance rules, by calculating two separate increments to be applied.

More specifically, if it takes N bytes for the cwnd to reach ssthresh, and `total_increment` $> N$, then it follows that, of the bytes of the acknowledgment, an amount equal to

$$\text{ACKed bytes} \cdot \frac{N}{\texttt{total\_increment}}$$

should be handled as per SS rules, whereas the rest, which is

$$\text{ACKed bytes} \cdot \frac{\texttt{total\_increment} - N}{\texttt{total\_increment}}$$

should be handled as per CA rules.

Processing the first subset of bytes according to SS can be achieved by simply setting the congestion window to ssthresh. Then, the amount of acknowledged bytes to process according to CA is derived by applying the reasoning shown above: `excess_increment` is equivalent to `total_increment` $- N$, and is then used to obtain `excess_bytes`, which in turn is processed according to CA rules to derive the second increment.

The function `picoquic_hybla_get_raw_ca_increment` applies congestion avoidance rules based on a given number of bytes, and returns the resulting cwnd increment. The returned value is fractional, since ρ is part of its calculation.

Once the cwnd increase is applied, if the cwnd has reached ssthresh, the algorithm transitions to the congestion avoidance state.

Congestion avoidance

Hybla's congestion avoidance phase is implemented as follows.

```
case picoquic_hybla_alg_congestion_avoidance:
default: {
    update_rho(hybla_state, path_x);

    double increment = picoquic_hybla_get_raw_ca_increment(hybla_state,
    ↪   path_x, ack_state->nb_bytes_acknowledged);

    uint64_t increment_int_part = floor(increment);
    double increment_frac_part = increment - increment_int_part;

    hybla_state->cwin += increment_int_part;

    hybla_state->increment_frac_sum += increment_frac_part;
    if (hybla_state->increment_frac_sum >= 1.0) {
        hybla_state->cwin += 1;
        hybla_state->increment_frac_sum -= 1.0;
    }
break;
```

As in Hybla's slow start phase, the `increment_frac_sum` variable is used to accumulate the fractional parts of each increase.

The function that calculates the CA increment is the following:

```
static double picoquic_hybla_get_raw_ca_increment(
picoquic_hybla_state_t* hybla_state, picoquic_path_t* path_x, uint64_t nb) {

    double rho2 = hybla_state->rho * hybla_state->rho;
    double increment = rho2 * nb * path_x->send_mtu / hybla_state->cwin;

    return increment;
}
```

In the NewReno congestion avoidance rules, the cwnd increase – in MSS units – is $1/\text{cwin}$. In the Picoquic implementation of NewReno, since the cwnd is in bytes, the increment is calculated as follows:

```
(ack_state->nb_bytes_acknowledged * path_x->send_mtu +
↪   nr_state->residual_ack) / hybla_state->cwin;
```

Where `nb_bytes_acknowledged` is the number of bytes acknowledged by the newly arrived ACK, `send_mtu` is the maximum size of a QUIC packet in bytes, `cwin` is the congestion window in bytes, and `residual_ack` is a variable used to accumulate the fractional parts from earlier calculations.

The reasoning behind this approach can be derived as follows:

$$\frac{\text{ACKed\_bytes} \cdot \text{MSS}}{\text{cwin}_{\text{bytes}}} = \frac{\text{ACKed\_bytes} \cdot \text{MSS}}{\text{cwin} \cdot \text{MSS}} = \frac{1}{\text{cwin}} \cdot \text{ACKed\_bytes}$$

Where $\text{cwin}_{\text{bytes}}$ is the congestion window in bytes. The expression shows that the formula used by the NewReno implementation is equivalent to the one in MSS units multiplied by the amount of bytes acknowledged.

In the Hybla implementation, the cwnd increase during congestion avoidance is the same as NewReno's, but with a multiplicative factor of ρ^2, as per the Hybla proposal.

Packet pacing

At the end of the notify function, the parameters of packet pacing are updated. This is done using the `picoquic_update_pacing_rate` function, which, in addition to the Picoquic context and a struct representing the current transport-level path, requires the parameters:

- `pacing_rate`, which is the desired transmission speed in bytes per second.
- `quantum`, which is the maximum amount of packets that can be sent at once.

```
uint64_t quantum = hybla_state->cwin / 4;
if (quantum < 2ull * path_x->send_mtu) {
    quantum = 2ull * path_x->send_mtu;
}
else if (quantum > 16ull * path_x->send_mtu) {
    quantum = 16ull * path_x->send_mtu;
}

picoquic_update_pacing_rate(
    cnx,
    path_x,
    (double)hybla_state->cwin / ((double)path_x->smoothed_rtt / 1000000),
    quantum
);
```

In Hybla, the pacing rate is set to be equal to the current congestion window (in bytes), divided by the current RTT estimate (in seconds), resulting in a uniform outflow of packets over time, without bursts. The quantum is set to be a quarter of the congestion window, and is clamped to a size between 2 and 16 packets, which is a common choice across the existing Picoquic congestion control implementations.

Other notification types

Of the remaining cases in the switch of the notify function:

- The cases `picoquic_congestion_notification_timeout`, `picoquic_congestion_notification_ecn_ec`, and `picoquic_congestion_notification_repeat` are handled similarly to NewReno, since there are no large differences between the NewReno and Hybla implementations in regard to the logic of the recovery phase.
- The case `picoquic_congestion_notification_spurious` occurs when a spurious repeat is detected, meaning that a packet that was thought to be lost is discovered to have actually been delivered. When this happens, both the NewReno and Hybla implementations will apply a mechanism to undo the changes to the congestion window made by the erroneously-triggered recovery process: the current congestion window, which was halved by the recovery process, is doubled in order to restore it to its previous value, and the state of the algorithm is set back to congestion avoidance.
- The case `picoquic_congestion_notification_rtt_measurement` is called each time that Picoquic obtains a new measurement of the path's RTT, which happens very frequently. In this section, both NewReno and Hybla enforce constraints on the minimum value of the congestion window. In Hybla, the minimum congestion window is set to

be that of NewReno, but multiplied by ρ, as specified in the Hybla proposal.

4.3 External parameters

The Hybla implementation includes two parameters: RTT_0 and `INITIAL_SSTHRESH`.

- To provide Hybla with its RTT_0 parameter, the option `-H` has been added to Picoquic's interface. The option requires a numeric value in milliseconds, which is then assigned to an internal variable within the Hybla implementation. As with the other Picoquic options, it can be passed to the provided testing software, such as `picoquicdemo`.
- Contrary to the RTT_0 parameter, `INITIAL_SSTHRESH` is not part of the original Hybla proposal. As such, it was decided to not implement it as a Picoquic option, but rather as an environment variable, obtained at runtime using the `getenv` C function.

Chapter 5 Validation testing and performance evaluation

5.1 Validation testing

Initial evaluation and debugging of the implemented Hybla congestion control was carried out using the visualization tool QVIS, which can display sequence diagrams of packet exchanges and graphs of relevant values over time, such as packets received, losses, acknowledgments, congestion window, in-flight packets, and more [QVIS].

To generate its visual representations, QVIS requires .qlog files as input [qlog]. These are files in a standard format that contain a variety of data pertaining to a connection, and they can be generated by a `picoquicdemo` endpoint using the `-q` option.

Initial testing was carried out by means of two virtual machines created through the VM management program Virtualbricks [Virtualbricks]. The program is equipped with channel emulation functionality, allowing for the customization of parameters such as RTT, loss rate, and bandwidth of channels. This allowed for a variety of initial tests, to check the correct functionality of the Hybla implementation step-by-step.

5.2 Performance evaluation

The work of this thesis was a prerequisite for a broad performance evaluation of the QUIC protocol in the context of LEO and GEO satellite networks. This evaluation is the main object of one of the companion theses of this project [Andreetti_2025]. For this reason, this section will be limited to a brief description of the testing methodology and the types of tests carried out, as a more extensive description can be found in the cited thesis.

Though logically distinct, the three theses that make up the project were carried out in parallel, with continuous exchanges of information. As a matter of fact, certain features of the Hybla implementation are inspired by the results of preliminary tests, such as the addition of the initial slow-start threshold limit, to avoid the excessive cwnd increase in scenarios with high

RTT (e.g. 500 ms) but low RTT_0 (e.g. 25 ms).

The performance evaluation was carried out by means of the Virtualbricks testbed shown in figure 5.1.

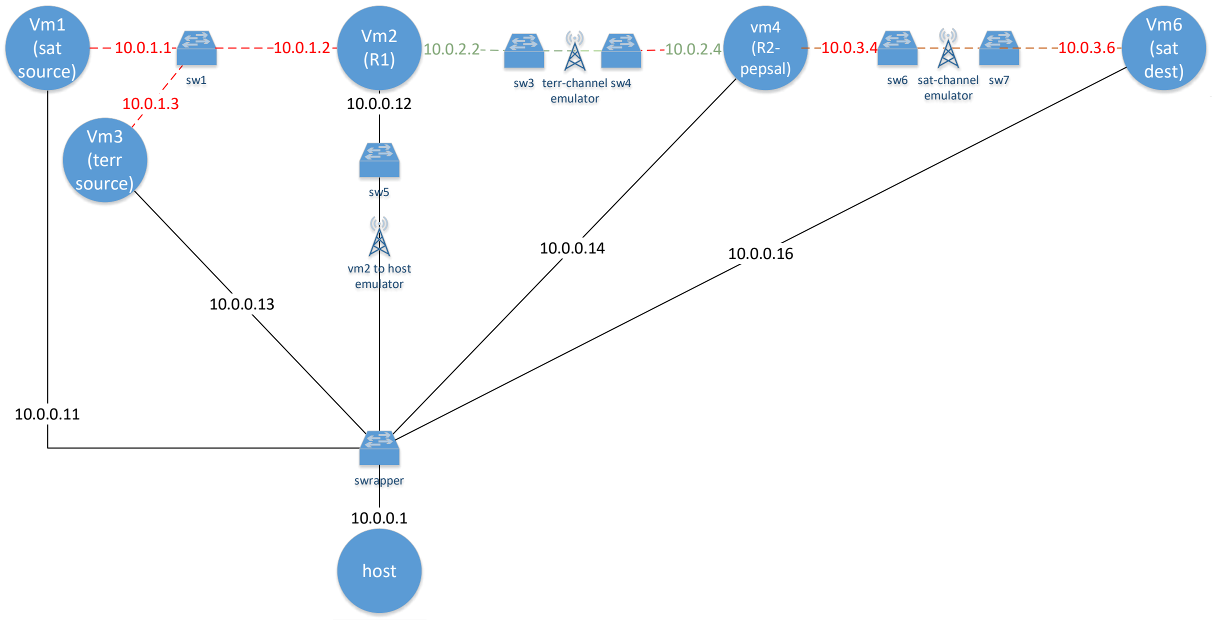

Figure 5.1: Virtualbricks testbed used in the final evaluation phase.

VM1 is used as a satellite sender, and VM3 is used as an additional terrestrial sender for tests that involve competing traffic; VM6 is used as the satellite receiver. VM4 acts as a both a satellite gateway and a terrestrial receiver. The link between VM2 and VM4 represents a terrestrial bottleneck, but it is not relevant in testing, so it is set to have an RTT of 0 ms, no losses, and no bandwidth constraints.

The link between VM4 and VM6 acts as a satellite link, and is customized through a channel emulator to have many different characteristics, which will be listed in the description of the tests.

The entire process from test execution to result collection was automated using bash scripts, executed on the host machine. Each script uses `ssh` commands on the VMs to configure channel emulators and start application instances (e.g. `picoquicdemo`, `iperf`); this is done many times, each time using a different combination of congestion control and channel characteristics. Each test execution produces log files, which are collected at the end of the script, and converted into a form that facilitates visualization.

On each test execution, the applications using Picoquic to communicate output log files on the VMs (both through Picoquic's own logging capabilities, as well as through custom logging functionality for certain values otherwise not logged). These files are then automatically downloaded on the host machine and converted into .csv files. Finally, the .csv data is injected in the sheets of custom .ods LibreOffice files, which are already set up to display the desired graphs.

The automated editing and manipulation of LibreOffice files was done in a separate Python script, through the use of the Pyuno python module [Pyuno], which provides access, in Python, to the UNO (Universal Network Objects) API used by both OpenOffice and LibreOffice to allow applications to interact programmatically with elements of their office suites.

Of the tests conducted by Luca Andreetti [Andreetti_2025], the ones of greater interests to this thesis are the ones regarding transmission between `picoquicdemo` instances. These tests are centered on the comparison of the performance of various congestion control algorithms, and result in graphs of the congestion window over time and goodput over time.

The tests involving `picoquicdemo` instances include a first set of tests that collect results regarding the standalone performance of various congestion control algorithms, in LEO and GEO scenarios, with a variety of channel conditions. Each test consists in the transmission of a constant stream of data – as much as possible – from one endpoint to another, using a different combination of the following parameters at each execution:

- Congestion control of the sender: NewReno, Hybla, Cubic, BBR.
- For the satellite link:
  - RTT: 30 ms, to simulate a LEO satellite link, and 500 ms, to simulate a GEO satellite link.
  - Data rate: 10, 25, 50, and 100 Mb/s.
  - Loss rate: 0.001%, 0.1%, 1% (and 10%, only in the GEO experiment).

Additionally, to evaluate fairness and friendliness of the QUIC Hybla implementation, the scripts execute the same tests again, but with two connections sharing the network concurrently, where one is always using QUIC Hybla. The process is repeated for each competitor listed below:

- QUIC with Hybla
- QUIC with NewReno
- QUIC with Cubic
- QUIC with BB
- TCP PEPsal (only in GEO experiments) – this is a type of PEP (Performance Enhancing Proxy) that helps mitigate the negative effects of long RTTs and losses, which are typical of satellite communications [PEPsal]. The core idea behind the technique is the splitting of high-RTT, end-to-end TCP connections, through the use of an intermediate node (PEP agent) that intercepts and retransmits TCP segments on a separate connection. This approach allows the use of appropriately-configured TCP versions on different sections of the network, e.g. the use of TCP Hybla exclusively on a satellite link.
  In the tests using PEPsal, TCP traffic is simulated using the bandwidth measuring tool Iperf [Iperf].

Both of the competing connections send a constant stream of data to the same destination endpoint, sharing the same satellite link, and thus the same bandwidth.

A thorough breakdown of the implementation of the scripts for testing and result collection can be found in Luca Andreetti's thesis.

5.3 Results

This section contains the analysis of a selection of results to demonstrate the performance of the implemented Hybla algorithm in the presence of GEO satellite links. The full set of results is available in [Andreetti_2025]

The graphs in figures 5.2 and 5.3 display the congestion windows over time of connections using Hybla and NewReno respectively, in a scenario analogous to a GEO satellite network. In both experiments, the satellite link has RTT 500 ms, bandwidth of 10 Mb/s, and packet error rate of 0.01%. Hybla's RTT_0 parameter is set to 25 ms, and the `INITIAL_SSTHRESH` parameter is set to 750 Kb. The results are from separate test executions, each with no competing connections. Both experiments lasted five minutes, but the graphs presented will be focused on the more relevant portions of the results.

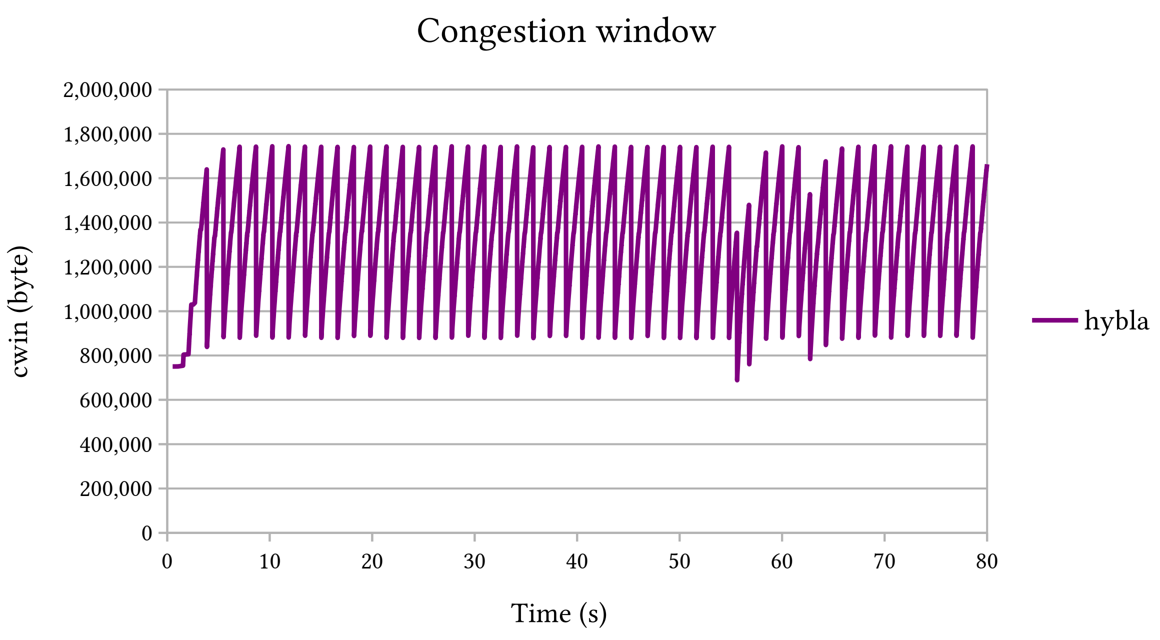

Figure 5.2: Cwnd of a `picoquicdemo` sender using Hybla, with RTT 500 ms, bandwidth 10 Mb/s, loss rate 0.01%, and RTT_0 parameter set to 25 ms.

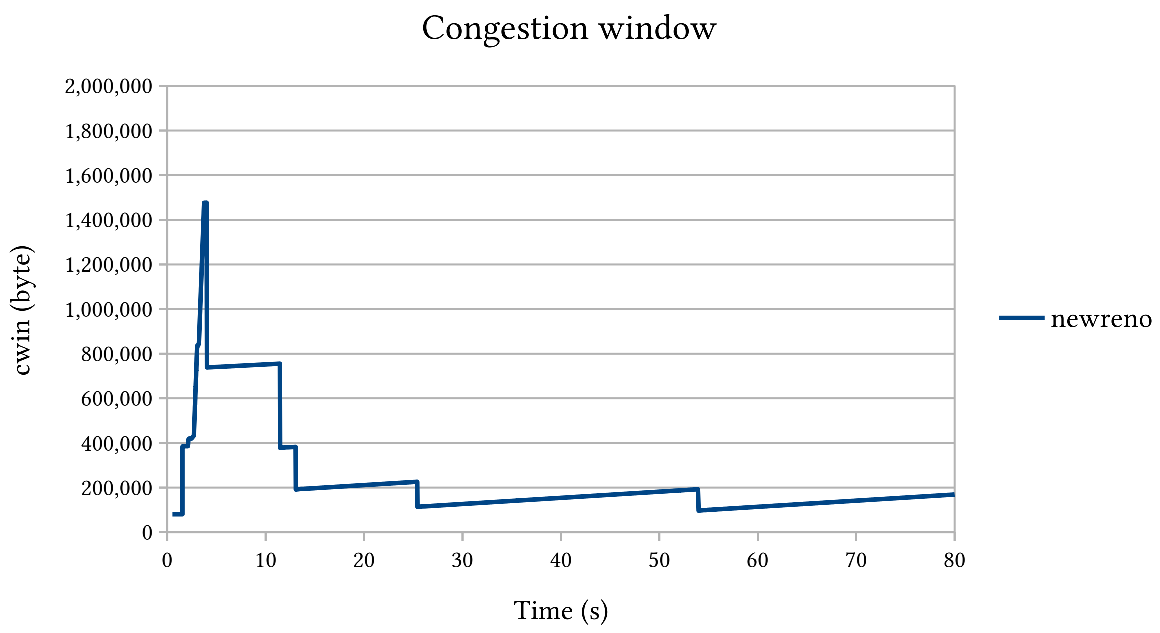

Figure 5.3: Cwnd of a `picoquicdemo` sender using NewReno, with RTT 500 ms, bandwidth 10 Mb/s, loss rate 0.01%.

The scenario of this experiment is analogous to those that Hybla is designed to handle. The value of Hybla's ρ parameter in this experiment is quite high, reaching a value of approximately $^{500\text{ ms}}/_{25\text{ ms}} = 20$.

The initial congestion window of Hybla is significantly greater than that of NewReno, as it is multiplied by the value of ρ, as per the original Hybla proposal.

In both graphs is visible an initial slow start phase, in which the congestion window grows exponentially with time. NewReno's slow start lasts until the first loss is detected, which as per QUIC's recovery logic results in the halving of the congestion window. After that, NewReno enters the congestion avoidance phase, in which the cwnd grows linearly.

As for Hybla, in this specific experiment it immediately transitions from slow start to congestion avoidance, due to the initial window being higher than the value of `INITIAL_SSTHRESH`. Hybla's CA results in a linear increase that is quite steep, and as a result frequently hits the bandwidth's limit, with losses that cause a halving of the congestion window. Additional losses, caused by the channel's packet error rate, cause further reductions of the cwnd, but they are quickly recovered from. Neither Hybla now NewReno are faced with a situation of persistent congestion, so the cwnd is never reset to its initial value, but rather only halved.

Overall, it is evident from the graph that Hybla's congestion window allows an amount of in-flight packets that occupies most of the available bandwidth.

On the other hand, NewReno's cwnd during CA increases extremely slowly. This is expected, as NewReno's algorithm for updating the cwnd is affected by the network's RTT. The amount of in-flight packets is heavily limited, which is bound to lead to poor bandwidth usage. The issue is exacerbated by the losses caused by the channel's PER, which take a long time to recover from.

Preliminary tests showed that high values of ρ, such as the one in this experiment, led to the Hybla cwnd spiking to extremely high values (in the order of 2^{20} bytes) in the slow start phase, consequently leading to a sizable burst of losses. However, by introducing the `INITIAL_SSTHRESH` parameter and implementing measures to manage excess bytes in the slow start phase, the issue was successfully mitigated.

The graphs in figures 5.4 and 5.5 show the goodputs (i.e. the rate at which data successfully reach the receiver) of Hybla and NewReno in the same experiments shown above. The value is averaged over a sliding window of 1 second (i.e. each point of the graph corresponds to the average of the samples in the last second).

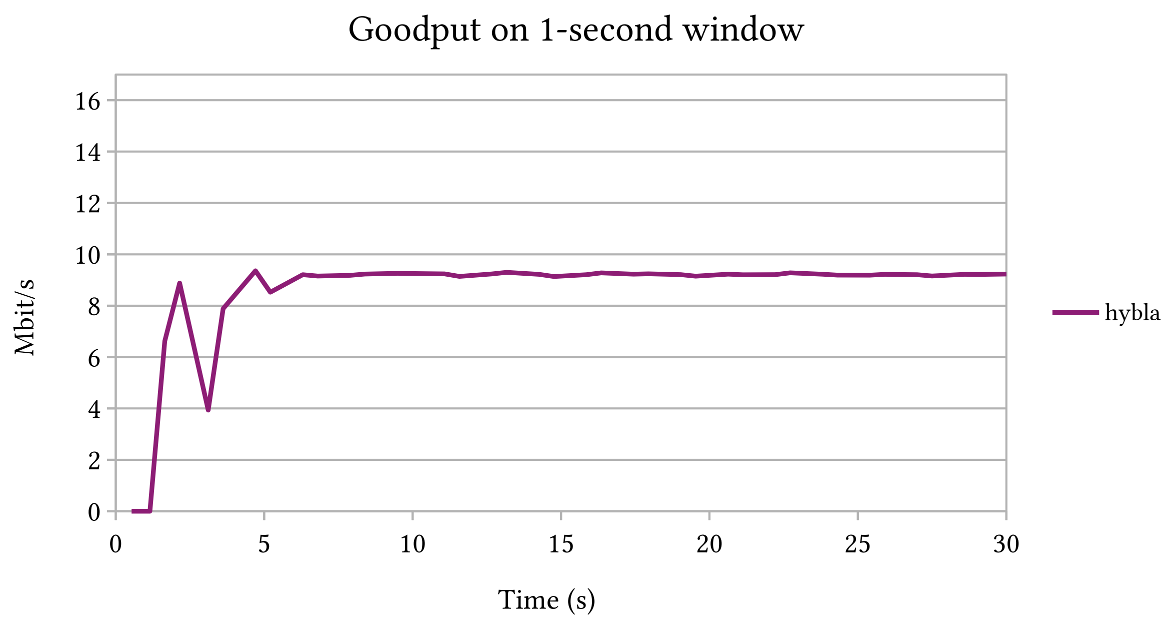

Figure 5.4: Goodput of a `picoquicdemo` sender using Hybla, with RTT 500 ms, bandwidth 10 Mb/s, loss rate 0.01%, and RTT$_0$ parameter set to 25 ms.

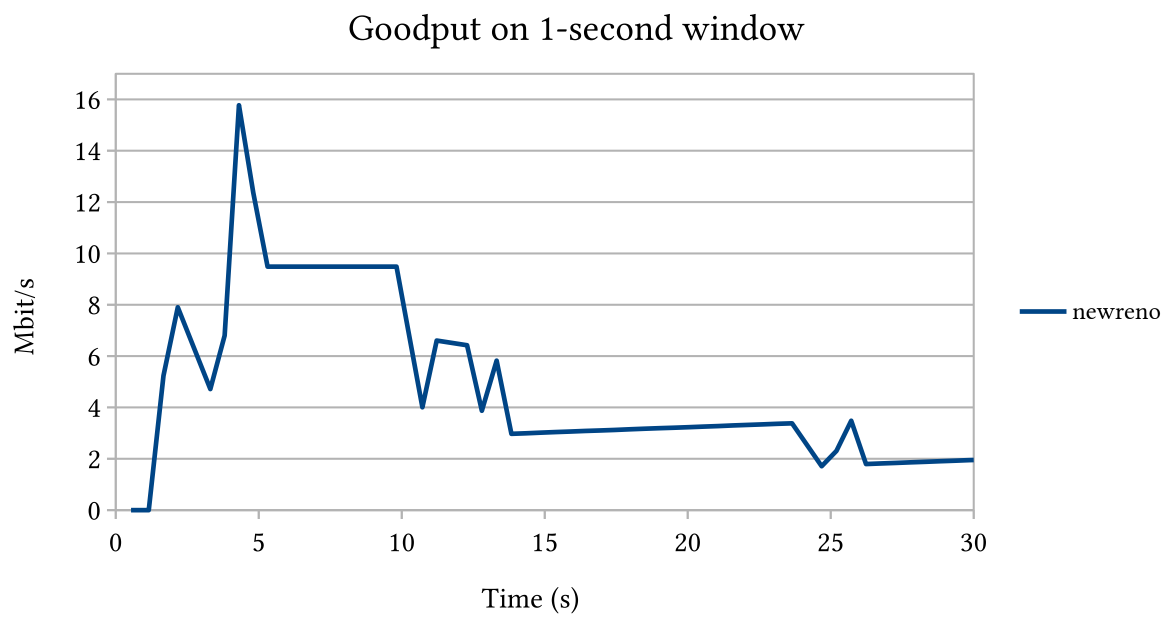

Figure 5.5: Goodput of a `picoquicdemo` sender using NewReno, with RTT 500 ms, bandwidth 10 Mb/s, loss rate 0.01%.

Once Hybla's goodput stabilizes, following the initial transitory period, it reaches a value above 9 Mb/s, with an average of 8.9 Mb/s, making use of nearly all of the available bandwidth. On the other hand, NewReno stabilizes at a very low value, at around 2 Mb/s, with an average of around 3.7 Mb/s.

As could be observed from the cwnd graphs, Hybla allows a higher amount of in-flight packets, meaning that its bandwidth usage is higher, which in turn implies a higher goodput. For NewReno, the opposite is true: the low amount of in-flight packets results in data being delivered at a very slow rate.

In the experiments with a RTT of 30 ms, modeled after a LEO satellite network, Hybla's performance is more modest – closer to that of NewReno – as the value of $\rho = \mathrm{RTT}/\mathrm{RTT}_0$ results closer to 1, specifically around 1.2.

Hybla's fairness and friendliness were evaluated based on the results of the tests with competing connections. Results show that QUIC Hybla is fair when competing against QUIC senders that are also using Hybla with the same RTT_0, and it is friendly when facing NewReno, BBR, and Cubic connections that have a RTT close to RTT_0. This is consistent with the fact that Hybla emulates a NewReno connection with RTT equal to RTT_0 – analogous behavior was also described in the test section of the original TCP Hybla proposal.

The complete set of graphs and observations, such as those regarding Hybla's performance in the presence of higher packet error rates, its performance when competing against various other connections, and its performance in the context of DTN, can be found in [Andreetti_2025].

Chapter 6 Conclusion

The objective of this thesis was to design and implement a QUIC version of the Hybla congestion control algorithm originally designed for TCP, as part of a broader project aimed at evaluating the performance of the QUIC transport protocol in the context of GEO and LEO satellite networks. The work was conducted at the German Aerospace Center (Deutsches Zentrum für Luft- und Raumfahrt, DLR), in collaboration with the University of Bologna.

Hybla was chosen as it was specifically designed to counteract the long RTTs typical of GEO satellite links, making it suitable for the scenarios of interest to the project.

After a comprehensive analysis and understanding of the QUIC specification and the original TCP Hybla proposal, Hybla has been successfully implemented within an existing QUIC implementation, Picoquic, and is currently in the process of being integrated in the official Picoquic repository.

Hybla's congestion window updating rules have been adapted to function within the Picoquic implementation, and the other features envisaged by Hybla, that were originally meant for TCP, have been examined and adapted to the QUIC environment.

As has been observed from the results of the performance evaluation, QUIC Hybla exhibits congestion window dynamics that are consistent with those of TCP Hybla, with slow start and congestion avoidance phases that function as expected. As a result, high-RTT connections are appropriately boosted, preventing the bandwidth starvation of RTT unfairness; the performance increase compared to NewReno becomes more significant the greater the connection's RTT, which is also consistent with the behavior of TCP Hybla.

The external parameters of Hybla, namely RTT_0 and `INITIAL_SSTHRESH`, have been implemented and successfully utilized in the testing phase.

In conclusion, the Hybla implementation for the QUIC protocol functions as intended, having undergone extensive validation and testing, successfully fulfilling the original aim of this work.

Bibliography

[Andreetti_2025] L. Andreetti, "QUIC performance evaluation in satellite networks: development of tools and result analysis", Master's thesis, University of Bologna, Mar. 2025.

[BBR] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-Based Congestion Control: Measuring bottleneck bandwidth and round-trip propagation time", in: *Queue* 14.5 (Oct. 2016), pp. 20–53, ISSN: 1542-7749, DOI: `10.1145/3012426.3022184`, URL: `http://dx.doi.org/10.1145/3012426.3022184`.

[Caini_2004] C. Caini and R. Firrincieli, "TCP Hybla: a TCP enhancement for heterogeneous networks", in: *International Journal of Satellite Communications and Networking* 22.5 (Aug. 2004), pp. 547–566, ISSN: 1542-0981, DOI: `10.1002/sat.799`, URL: `http://dx.doi.org/10.1002/sat.799`.

[FAST] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "FAST TCP: Motivation, Architecture, Algorithms, Performance", in: (), URL: `https://web.archive.org/web/20060906104225/http://netlab.caltech.edu/pub/papers/FAST-ToN-final-060209.pdf`.

[Hybla_Linux] *Source code of the TCP Hybla implementation in Linux*, URL: `https://github.com/torvalds/linux/blob/master/net/ipv4/tcp_hybla.c`.

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

[Moffa_2025] M. Moffa, "Design and implementation of a DTN Convergence Layer Adapter based on the QUIC protocol", Master's thesis, University of Bologna, Mar. 2025.

[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, IEEE, pp. 2607–2611, DOI: `10.1109/vetecs.2006.1683339`, URL: `http://dx.doi.org/10.1109/VETECS.2006.1683339`.

[Picoquic] *Official Picoquic source code*, URL: `https://github.com/private-octopus/picoquic`.

[Pyuno] *Pyuno Python module*, URL: `https://wiki.openoffice.org/wiki/Python`.

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

[quicperf] N. 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] *QUIC and HTTP/3 visualization toolsuite*, URL: `https://qvis.quictools.info`.

[RFC3168] S. Floyd, D. K. K. Ramakrishnan, and D. L. Black, *The Addition of Explicit Congestion Notification (ECN) to IP*, RFC 3168, Sept. 2001, DOI: `10.17487/RFC3168`, URL: `https://www.rfc-editor.org/info/rfc3168`.

[RFC5681] E. Blanton, D. V. Paxson, and M. Allman, *TCP Congestion Control*, RFC 5681, Sept. 2009, DOI: `10.17487/RFC5681`, URL: `https://www.rfc-editor.org/info/rfc5681`.

[RFC5827] J. Blanton, P. Hurtig, U. Ayesta, K. Avrachenkov, and M. Allman, *Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)*, RFC 5827, Apr. 2010, DOI: `10.17487/RFC5827`, URL: `https://www.rfc-editor.org/info/rfc5827`.

[RFC6298] M. Sargent, J. Chu, D. V. Paxson, and M. Allman, *Computing TCP's Retransmission Timer*, RFC 6298, June 2011, DOI: `10.17487/RFC6298`, URL: `https://www.rfc-editor.org/info/rfc6298`.

[RFC6582] A. Gurtov, T. Henderson, S. Floyd, and Y. Nishida, *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`.

[RFC8312] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger, *CUBIC for Fast Long-Distance Networks*, RFC 8312, Feb. 2018, DOI: `10.17487/RFC8312`, URL: `https://www.rfc-editor.org/info/rfc8312`.

[RFC8985] Y. Cheng, N. Cardwell, N. Dukkipati, and P. Jha, *The RACK-TLP Loss Detection Algorithm for TCP*, RFC 8985, Feb. 2021, DOI: `10.17487/RFC8985`, URL: `https://www.rfc-editor.org/info/rfc8985`.

[RFC8999] M. Thomson, *Version-Independent Properties of QUIC*, RFC 8999, May 2021, DOI: `10.17487/RFC8999`, URL: `https://www.rfc-editor.org/info/rfc8999`.

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

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

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

[RFC9221] T. Pauly, E. Kinnear, and D. Schinazi, *An Unreliable Datagram Extension to QUIC*, RFC 9221, Mar. 2022, DOI: `10.17487/RFC9221`, URL: `https://www.rfc-editor.org/info/rfc9221`.

[RFC9369] M. Duke, *QUIC Version 2*, RFC 9369, May 2023, DOI: `10.17487/RFC9369`, URL: `https://www.rfc-editor.org/info/rfc9369`.

[Virtualbricks] P. Apollonio, C. Caini, M. Giusti, and D. Lacamera, "Virtualbricks for DTN Satellite Communications Research and Education", in: *Personal Satellite Services. Next-Generation Satellite Networking and Communication Systems*, ed. by I. Bisio, Cham: Springer International Publishing, 2016, pp. 76–88, ISBN: 978-3-319-47081-8.

Acknowledgments

I would like to express my gratitude to my family for their continuous support throughout the process of working on this thesis, as well as their support throughout my life.

I also extend my appreciation to my friends Mattia Moffa and Luca Andreetti, who have developed their theses in parallel to mine, and with whom I have shared not only the last six months living in Munich but also the five years of my studies at the University of Bologna.

Finally, I am thankful to the people working at the Institute of Communication and Navigation of the German Aerospace Center, for their friendliness, insights and support during my research. Among them, I am especially thankful to my co-supervisor Tomaso de Cola for his guidance and assistance in the completion this work.