FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Radek Zikmund

QUIC Protocol Implementation for .NET

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	Mgr. Pavel Ježek, Ph.D.
Study programme:	Computer Science
Study branch:	Software and Data Engineering

Prague 2021

I want to thank my family, friends, and everyone who supported me and helped me during my studies. Especially, I would like to thank my supervisor Pavel Ježek for his time, patience, and advice he gave me. Without his guidance, this thesis would not reach the level of quality I believe it has. I also greatly appreciate the support from the Microsoft .NET team and would like to explicitly thank Karel Zikmund for his counseling, insight, and help with tuning the thesis topic, and Marie Píchová and Geoffery Kizer for their valuable help and feedback.

Title: QUIC Protocol Implementation for .NET

Author: Bc. Radek Zikmund

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract: QUIC is a general-purpose transport layer network protocol proposed as the replacement for TCP and TLS in HTTP/3. QUIC is based on UDP and provides always-encrypted connections able to transmit multiple streams of data in parallel. Compared to TCP, QUIC promises lower latency, better congestion control flexibility, and a solution to head-of-line blocking occurring in multiplexed HTTP/2 connections.

The latest release of .NET — .NET 5 — has shipped with experimental support for QUIC based on the MsQuic C library. However, when implementing new features in standard .NET libraries, purely .NET implementations are preferable to adding dependencies on native libraries because .NET implementations offer better maintainability and — in some cases — even better performance. This thesis explores the viability of a purely C# QUIC implementation as a future replacement for .NET 6 or later release.

This thesis's result is a fork of the official .NET runtime repository with partial C# implementation of the QUIC protocol. We implemented a subset of the QUIC specification, which is sufficient for a basic performance evaluation. As part of the thesis, we have benchmarked the throughput and latency of our and the MsQuic-based QUIC implementation and compared them to TCP's performance in two environments: LAN and simulated cellular network. While the benchmarking results show that our implementation is slower than the MsQuic-based one, we identified the primary performance limiting factors in our implementation and suggested the course for future development.

Keywords: QUIC .NET network protocol

Contents

1. Introduction

The internet, as we know it today, heavily relies on the use of the HTTP protocol. Is it used not only by web browsers for interacting with web applications, but it may also be used at the server side for communication between individual nodes in a cluster. Such use of HTTP is commonly seen in, e.g., microservices architecture. Finally, HTTP is also used as a transport medium for RESTful web APIs and technologies such as gRPC and GraphQL.

The latest version of the HTTP protocol is HTTP/2, published in 2015 [1]. HTTP/2 improved on its predecessor HTTP/1.1 [2] by introducing request header compression, request multiplexing over a single TCP connection, and server-push features. These features led to reduced loading times of web pages and generally improved the efficiency of the web [3].

1.1 Remaining Performance Issues in HTTP/2

HTTP/2, however, does not solve all the performance issues HTTP/1.1 had. HTTP/2 introduced new features and changes only to the topmost layer of the HTTP stack, and these changes could not address the performance limiting phenomenons caused by the layers underneath. For the context of this thesis, the following two performance problems are the most relevant.

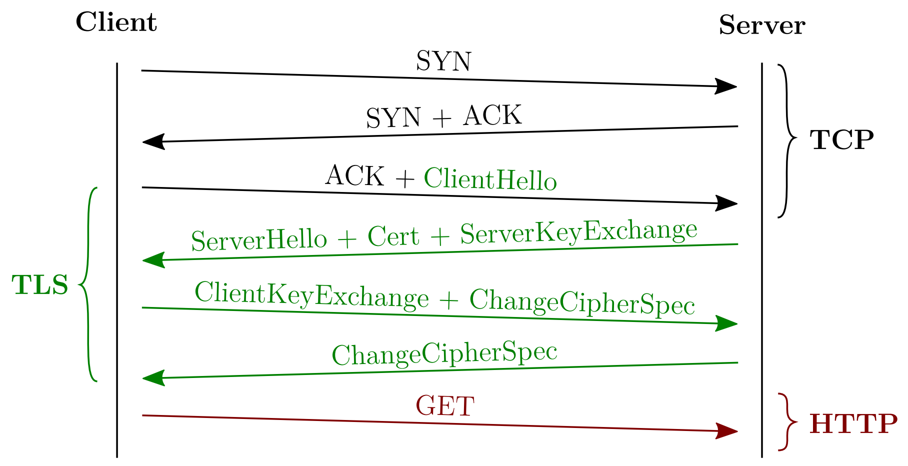
Head-of-Line Blocking

Request multiplexing was introduced in HTTP/2 to reduce the number of web servers' resources required to serve requests made by web browsers. Because modern web pages are composed of many parts (HTML, images, Javascript files, CSS style sheets), web browsers have to make multiple HTTP requests to load the entire page. HTTP/1.1 creates multiple independent HTTP connections to download individual parts of the webpage, one connection per HTTP request. HTTP/2, on the other hand, can make multiple HTTP requests in parallel in a single connection. Although this change improved HTTP's performance, its current design is limited by a phenomenon known as *head-of-line blocking*.

The individual HTTP frames which make up the HTTP requests and responses are interleaved and transferred over TCP as a single stream of data. When a TCP packet carrying a part of this stream is lost, delivery of the data in all following packets is delayed until the lost packet is retransmitted. This will cause a delay in all HTTP requests currently in progress, even those requests whose data were not carried by the lost packet. More information on head-of-line blocking can be found, e.g., on the dedicated Wikipedia page [4].

HTTPS Connection Establishment Latency

HTTPS [5] is an extension to HTTP, which makes the connection encrypted by inserting the TLS protocol layer between HTTP and TCP. Therefore, establishing an HTTPS connection requires establishing a TCP connection first — performing the three-way handshake — and then performing another separate handshake for the TLS layer. As can be seen on the illustration in Figure 1.1, three round trips are needed at minimum before the HTTP request can be even sent.



Client
Server
SYN
SYN + ACK
ACK + ClientHello
ServerHello + Cert + ServerKeyExchange
ClientKeyExchange + ChangeCipherSpec
ChangeCipherSpec
GET
TCP
TLS
HTTP

Figure 1.1: Packets sent during HTTPS connection establishment

More and more websites enforce the use of HTTPS in order to protect the privacy of their users. In August 2020, 96 out of the top 100 viewed websites actively redirected to HTTPS, and more than 75% of all network traffic from the Chrome web browser used HTTPS [6]. Even server to server communication in the cloud is trending towards HTTPS to protect against partially compromised networks. With HTTPS becoming the norm, almost all connections suffer from the increased latency caused by the additional TLS handshake.

1.2 HTTP/3 and QUIC

The next version of the HTTP protocol — HTTP/3 [7] — addresses the issues mentioned above by replacing the TCP and TLS layers with a brand new UDP-based protocol named QUIC[1]. The QUIC protocol features allow moving multiplexing capability from the application layer into the transport layer of the HTTP protocol stack. The responsibilities and relationships between protocols on the HTTP/2 and HTTP/3 stacks are illustrated in Figure 1.2.

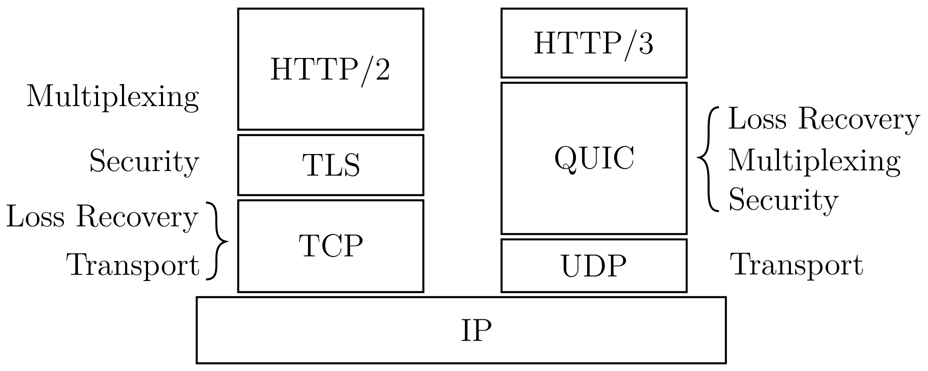


Figure 1.2: Comparison between HTTP/2 and HTTP/3 protocol stacks

[1]Originally intended as the acronym for Quick UDP Internet Connections. However, it has been changed to be the protocol's actual name during the standardization process

Although QUIC development is tied with that of HTTP/3, it is designed as a general-purpose transport layer protocol that can be used for other application-layer protocols. The following points summarize the main improvements of QUIC over TCP+TLS:

- *Stream multiplexing*: QUIC provides an abstraction of multiple streams of data multiplexed on a single connection. Moreover, because QUIC itself also implements loss detection and recovery, packet loss can be managed in a way that reduces the scope of the head-of-line blocking problem described above to only the streams whose data need to be retransmitted.

- *Faster connection establishment*: Internally, QUIC also performs the TLS handshake, but it does so in parallel with the base protocol's handshake. The combined handshake requires fewer round-trips and is faster than the combination of TCP and TLS.

  QUIC also supports opt-in Zero round trip time resumption (0-RTT) from TLS 1.3. The 0-RTT mode of operation allows a client to cache some session information, allowing it to send application layer data with the first packet in future connections to the same server. 0-RTT effectively reduces the connection establishment latency by another round trip but makes applications vulnerable to repeat attacks. A more detailed description of 0-RTT can be found online, e.g., on Cloudflare's blog [8].

- *Always encrypted*: TLS handshake is a mandatory part of connection establishment, and encryption, therefore, cannot be turned off. This makes QUIC secure-by-default.

- *Separating connection identity from peer's IP address*: QUIC protocol does not use peers' IP addresses to identify connections but instead uses Connection IDs, which are 8 to 20-byte sequences negotiated during connection establishment.

  This makes QUIC very attractive for mobile devices, which can change IP addresses due to switching between Wi-Fi and cellular data network. In TCP and — by extension — HTTP/2, the existing connection must be terminated, and a new connection established from the new IP address. QUIC, on the other hand, can migrate the connections in a way that is transparent to the application layer.

  This feature also enables QUIC extensions like Multipath QUIC [9], which allows simultaneous use of multiple network interfaces for a single connection to achieve greater throughput.

As of August 2020, the specifications of HTTP/3 and QUIC are still at the draft stage, but the standardization process is believed to be very close to complete. There are already multiple implementations of QUIC being developed based on the draft versions of the standard. These implementations are backed by big companies such as Google, Cloudflare, Facebook, and Microsoft.

Experiments with these implementations allowed early performance comparison between HTTP/3 and HTTP/2, yielding promising results. In 2015, the Chromium team's experimental implementation showed a 3% improvement in mean page load time and 30% fewer rebuffer events when watching YouTube videos [10]. Cloudflare launched preliminary support for HTTP/3 in April 2020 and has measured a 12.4% decrease in the *time to the first byte* metric [11], which is consistent with the QUICs promise of reduced latency. In measurements done by Orange Labs, QUIC protocol significantly outperforms TCP in unstable networks such as wireless mobile networks [12].

1.3 Support for QUIC in .NET

Microsoft's .NET development team has long-term plans to provide full support for QUIC and HTTP/3 in .NET. Support for HTTP/3 should be completely transparent to users because the implementation of `HttpClient` automatically chooses the best available HTTP version [13]. However, since QUIC can be used to build other protocols than HTTP/3, its implementation will be exposed via public classes residing most likely inside the `System.Net.Quic` namespace.

The QUIC support has been initially intended for the .NET 5 release planned for November 2020. However, it turned out that the standardization process would not be completed in time for QUIC to be implemented for the release. .NET 5, therefore, ships with only a preview (non-production ready) HTTP/3 and QUIC support. A production-ready HTTP/3 and QUIC implementation was postponed until .NET 6 release.

Existing QUIC Implementation in .NET

The current work-in-progress support for QUIC in .NET 5 is a wrapper around the *MsQuic* library [14], which is a C implementation of QUIC developed by Microsoft. The *MsQuic* library was designed for high-performance scenarios and has been recently made open-source.

The decision to use *MsQuic* as the QUIC protocol implementation is not final. There are compelling arguments for implementing the QUIC protocol in managed .NET code — and, more specifically, in C# — for the production release.

The existing QUIC implementation in .NET uses a layer of indirection which allows multiple implementations to exist side by side. Furthermore, it is even possible to choose which QUIC implementation should be used at runtime. This fact can be used to implement benchmarks comparing the performance of available QUIC implementations.

Motivation for Implementing QUIC in Managed Code

Code written in ahead-of-time compiled languages such as C or C++ (referred to as *native code*) is likely to be faster than code written in .NET languages (referred to as *managed code*), which rely on the just-in-time compilation. However, there are other aspects than raw performance to be considered when deciding to use native libraries such as *MsQuic*:

- *Cross-platform compatibility/availability*: The .NET platform officially supports multiple versions of Windows, macOS, and several Linux distributions. If the native library does not support all these platforms, then the implementation must use an alternative library on other platforms, introducing more complexity into the codebase and possible incompatibilities between platforms.

- *Support for different library versions*: Currently, no native libraries that managed .NET libraries depend on are part of the .NET distribution itself. Instead, .NET runtime expects that the library is already installed on the target machine and can be dynamically loaded. There is no way to enforce a specific version of the library, which means that the .NET code must work correctly with multiple versions of the native library.

- *Maintainability*: Maintenance of the interop code requires the developer to read and understand the language in which the native library is written. Debugging the code around the language boundaries can be difficult if the necessary tooling for mixed-language debugging is not available.

- *Library's API*: One of the aspects which greatly influences the resulting performance is the similarity between the public API of the .NET library and the API of the native library. Some aspects are the following:

  - Event-based (using callbacks) vs. method based
  - Blocking vs. non-blocking
  - Synchronous vs. asynchronous
  - Which side allocates memory buffers

  Managed code needs to translate the differences between APIs, which may result in noticeable performance overhead and even negate the performance gained from the native library's use.

- *Overhead of interop calls*: When execution transitions between managed and unmanaged code, the runtime must ensure the correct behavior of exceptions and garbage collector. The cost of the so-called managed-to-native transition is relatively small. However, it may be noticeable when the transition occurs very frequently.

- *Future development*: Exposing new features from newer versions of the native library can be problematic because the code needs to work with older versions of the library as well.

In the past, the .NET development team has encountered multiple problems with the `libcurl` [15] library, which was used to implement HTTP request handling on macOS and Linux. Different Linux distributions contained different versions of the `libcurl` library and, therefore, supported different features and had different bugs. The .NET development team had to expend many resources to make sure the managed code written by other .NET developers behaved consistently regardless of which `libcurl` version was present.

Starting with .NET Core 2.1, the default implementation of HTTP request handling does not rely on native libraries like `libcurl`. Instead, the functionality has been rewritten in pure managed code on top of `Socket` API. This implementation offered better performance and consistent behavior across all .NET platforms [16].

1.4 Goals of this Thesis

The reasons mentioned in the previous section motivated Microsoft developers to consider implementing QUIC in purely managed C# code. Before the final decision on which implementation of QUIC will be used, it is necessary to investigate the performance characteristics of managed QUIC implementation. This thesis seeks to create a partial implementation of QUIC whose performance can be analyzed and — if the managed implementation approach is found viable — can form the basis of the production QUIC implementation in future .NET versions.

Because this thesis's work can potentially become part of future .NET 6 release, the QUIC implementation development will occur inside a development branch of the .NET runtime repository [17]. The result should be a compilable branch of the runtime, which uses the managed QUIC implementation instead of the *`MsQuic`*-based one.

We have mentioned that the .NET QUIC API design was interrupted early. Although the current version does not expose all QUIC protocol features, it is sufficient for evaluating our implementation. This thesis will, therefore, avoid making any modifications to the API.

Implementing the full QUIC specification is outside the scope of a master thesis. Fortunately, to evaluate the managed QUIC implementation's performance, many parts of the specification can be omitted without affecting the core transport functionality. Therefore, this thesis will focus mainly on the functionality needed to reliably transport data between two endpoints and leave out advanced features such as connection migration.

Because we do not expect readers to be thoroughly familiar with the QUIC protocol specification and all its features, we will present an overview of the protocol in chapter 2 and defer the selection of the protocol features we will implement in this thesis to the beginning of the analysis in chapter 3. However, the implementation design should be such that the rest of the specification can be implemented in the future.

It would make sense to evaluate our QUIC implementation's functionality by providing a partial implementation of HTTP/3 as well. However, even supporting the most straightforward GET requests would be too complicated. Instead, we will implement a testing application in which the server will simply echo all data back to the client.

We briefly mentioned that the current QUIC implementation in .NET uses a layer of indirection which allows runtime selection of the QUIC implementation to be used. As an optional goal, this thesis will create a small benchmarking application and use it to to compare the performance of the two QUIC implementations. A small step from there would allow us to also compare the two QUIC implementations' performance with that of TCP+TLS-based `SslStream`.

Summary of the Goals

The following list summarizes the goals of this thesis presented in the previous subsection.

1. Select a sufficient subset of QUIC specification needed to support the most basic data transfer and implement it inside .NET runtime codebase.

2. Allow switching between the new managed implementation and the existing *`MsQuic`*-based one.

3. Evaluate the managed QUIC implementation by using it to implement a simple client-server echo application.

4. *(optional)* Try to compare the performance of the new implementation with the previous *`MsQuic`*-based one and with TCP+TLS-based `SslStream`.

2. QUIC Protocol

This chapter is intended to summarize the QUIC protocol specification and provide sufficient knowledge about the protocol needed for designing our implementation. This text is based on version 27 of the draft specification documents from February 2020, more specifically on the documents describing the core transport protocol [18], TLS integration [19], and congestion control mechanism [20]. Readers familiar with these documents may skip this chapter.

We will start this chapter by first providing a high-level overview of QUIC and then providing a more detailed description of the individual parts of the protocol.

2.1 Introduction

QUIC protocol provides reliable and secure transport of multiple streams of data over a single connection. QUIC provides the following services:

- Stream multiplexing
- Stream and connection-level flow control
- Low-latency connection establishment
- Connection migration and resilience to NAT rebinding

QUIC is implemented on top of UDP, which provides only unreliable transfer of datagrams. Therefore, in addition to stream multiplexing, QUIC also implements loss recovery, congestion control, transport security, and other features known from TCP or TLS protocols.

2.1.1 Basic Concepts

Throughout this chapter, and later when analyzing the QUIC protocol specification, this text will use several terms in a very specific meaning. This meaning is the same as defined in the main specification document [18, Section 1.2].

QUIC packet
: A complete processable unit of QUIC that is transported using UDP datagrams. Multiple QUIC packets can be encapsulated in a single UDP datagram but a single QUIC packet may not be split into multiple UDP datagrams.

Out-of-order packet
: A packet that does not arrive directly after the packet that was sent before it. A packet can arrive out of order if it was delayed, if earlier packets were lost or delayed, or if the sender intentionally skipped a packet number.

Endpoint
: An entity that can participate in a QUIC connection by generating, receiving, and processing QUIC packets. There are only two types of endpoints in QUIC: client and server.

Client
The endpoint initiating a QUIC connection.

Server
The endpoint accepting incoming QUIC connections.

Address
When used without qualification, an address is a tuple of IP version, IP address, and UDP port number.

Network path
A pair of two network address — one for each endpoint — used to exchange QUIC packets in a connection. Throughout its lifetime, a QUIC connection can change to a different network-path, e.g., by a process called connection migration.

Connection ID
An opaque identifier that is used to identify a QUIC connection at an endpoint. Each endpoint sets a value for its peer to include in packets sent towards the endpoint.

Stream
A unidirectional or bidirectional channel of ordered bytes within a QUIC connection. A QUIC connection can carry multiple streams simultaneously.

Application
An entity that uses QUIC to send and receive data. This can mean, e.g., a higher-level protocol built on top of QUIC or an actual application which uses QUIC directly.

2.1.2 Notational Conventions

This chapter contains graphical diagrams of several elements, such as QUIC packets. In those schematics, individual fields include length information as follows:

- *X (A)*: Indicates that X is A bits long.
- *X (i)*: Literal *i* indicates that X is encoded using the QUIC's variable-length encoding. This encoding is described in section 2.1.3.
- *X (A..B)*: Indicates that X can be any length from A to B. The actual length is generally stored in a different field.
- *X (..)*: Indicates that X can be any length. The actual length is generally stored in a different field.
- *X (A) = C*: Indicates that X has a fixed value C.
- *X (A) ...*: Indicates that X is repeated zero or more times and each instance is A bits long.

2.1.3 Wire Encoding

The process of encoding QUIC packets to be sent via the network is optimized for size. There are two nontrivial encodings used: variable-length integer encoding and packet number encoding. All values sent over the network are encoded in big-endian order.

Variable-Length Integer Encoding

Almost all numeric values stored in QUIC packets are encoded using a *variable-length integer encoding*. This encoding uses the first byte's two most significant bits to encode whether the value is encoded as 1, 2, 4, or 8-byte integer. This encoding supports only positive numbers. The ranges available for individual encoding lengths are listed in Table 2.1.

Table 2.1: Variable-length integer encoding lengths

Most significant bits	Encoding length (B)	Maximum value
00	1	63
01	2	16 383
10	4	1 073 741 824
11	8	$2^{62} - 1$

In order to optimize the size of QUIC packets, the implementations are encouraged to always choose the shortest encoding necessary to represent the given number.

Packet Number Encoding

Packets in QUIC are sequentially numbered, and, therefore, packet numbers received by an endpoint form a mostly ascending sequence with occasional reordering or gaps due to network unreliability. This causes the packet number of a newly received packet to be close to that of the previously received packet. QUIC leverages this fact by sending only the lower bytes of the packet number. The number of bytes sent varies between one to four bytes and is always chosen so that the receiver can reconstruct the original packet number using the previously received packet numbers and the bytes from the arriving packet.

When determining how many bytes of the packet number need to be included in the packet, the sender uses the highest acknowledged packet number. The encoding length must use enough bytes to be able to represent at least twice the difference between the current packet number and the highest acknowledged number.

As an example, consider the following situation: sender prepares to send packet 0x4417 and its highest acknowledged packet number is 0x43a0. The difference between the two numbers is 0x77 which is smaller than 128 and, therefore, requires only the least significant byte (0x17) of the packet number to be sent in the packet. Suppose that by the time the packet arrives at the receiver, the highest packet number it previously received was 0x43f5. Based on the encoded packet number length, the deconstructed packet number must be from the range

0x4375–0x4475. The only number from this range that matches the least significant byte sent in the packet is 0x4417 which is the correct packet number.

2.2 QUIC Packets

In QUIC connection, endpoints exchange *QUIC packets* enclosed in UDP datagrams. A single UDP datagram can contain multiple packets, although in most cases, they contain only one. Packing multiple QUIC packets into a single UDP datagram is called *coalescing*. A single QUIC packet cannot span multiple UDP datagrams. QUIC uses a total of six different packet types:

1. *Version Negotiation*: sent by the server when the client tries to establish a connection with QUIC version that is not supported by the server;

2. *Retry*: optionally used by servers for Client Address Validation when establishing new connections, address validation is described in section 2.8.3;

3. *Initial*: used to initiate a new connection and exchange the initial information;

4. *Handshake*: used during the connection handshake;

5. *0-RTT*: when TLS 1.3 0-RTT mode of operation is enabled, 0-RTT packets carry *early data* — application data that is sent before the TLS handshake is complete in order to reduce latency; and

6. *1-RTT*: main packet type used throughout the lifetime of QUIC connection. 1-RTT and 0-RTT packets are the only packet types that carry application data.

Version Negotiation and Retry packets are sent as one-time responses in particular scenarios and are not individually numbered. QUIC is designed to let the server send these packets without maintaining any state for the connection. These packets are not used in an established connection.

Packets of the other types — Initial, Handshake, 0-RTT, and 1-RTT — are individually numbered. The packet numbers do not form a single sequence, as is the case with TCP packets. Instead, QUIC organizes these packets into three separate *packet number spaces* and uses a separate sequence of numbers for each packet number space. These packet number spaces are:

1. *Initial*: Used for initiating new connections and exchanging initial cryptographic information. Contains only Initial packets.

2. *Handshake*: Used during the connection handshake process. Contains only Handshake packets.

3. *Application*: Used throughout the lifetime of the connection to transfer application data. This packet space contains both 0-RTT and 1-RTT packets.

In addition to being numbered separately, packets from each packet number space are processed entirely independently of the other packet number spaces. For example, Initial packets can be acknowledged only by another Initial packet. The four packet types also use different keys for encryption and offer increasingly stronger level of protection, with Initial packets being the least secure and 1-RTT packets the most secure. In their payload, the Initial, Handshake, 0-RTT, and 1-RTT packets carry *QUIC Frames* which are low-level protocol messages carrying, e.g., acknowledgments and parts of streams sent by the application.

Each packet type has a header and a payload. The packet header contains information such as Source Connection ID (SCID) and Destination Connection ID (DCID) which are Connection IDs used to match the packets to the correct receiving connection. QUIC optimizes the packet encoding to maximize the amount of application data sent in a single UDP datagram. Therefore, two types of packet headers exist. 1-RTT packets use a *short header*, and all other packet types use the *long header*. The long header contains fields like the protocol version identifier which are relevant only during connection establishment. The short header reduces the packet size by including only the DCID.

The type of the header is determined by the most significant bit of the first byte in the packet — the *Header Form* bit. The second most significant bit — the *Fixed Bit* — is always set to 1 in a valid QUIC packet. The following sections describe the structure of the rest of the packet for individual packet types.

2.2.1 Long Packet Header

Long headers contain information that is necessary for the connection establishment. It includes both the Source and Destination Connection IDs. It also contains *Version* and *Packet Type* fields. A few leftover bits of the header are reserved for type-specific information. Figure 2.1 illustrates the structure of the long header.

H=1	F=1	T (2)	Type Specific Bits (4)
Version (32)			
Destination Connection ID Length (8)			
Destination Connection ID (..)			
Source Connection ID Length (8)			
Source Connection ID (..)			

H = Header Form, F = Fixed Bit, T = Packet Type

Figure 2.1: Long packet header structure

The semantics of the individual fields of the long header is as follows:

Header Form (H)

Used to distinguish between packets with so-called *long header* and *short header* format. In the case of the long header packets. The Header Form bit is set to 1.

Fixed Bit (F)
A bit that is always set to 1 in a valid QUIC packet.

Packet Type (T)
Discriminator of the packet type. the possible values are listed in Table 2.2.

Table 2.2: Values of the Packet Type field in long packet header

Packet Type	Value
Initial	00
0-RTT	01
Handshake	10
Retry	11

Version
Indicates which version of QUIC is in use. The location of this field will be the same across all QUIC versions. However, the structure of the rest of the packet may be different in future versions of QUIC.

Destination Connection ID Length
Length of the Destination Connection ID field.

Destination Connection ID
The Connection ID issued by the recipient of the packet.

Source Connection ID Length
Length of the Source Connection ID field.

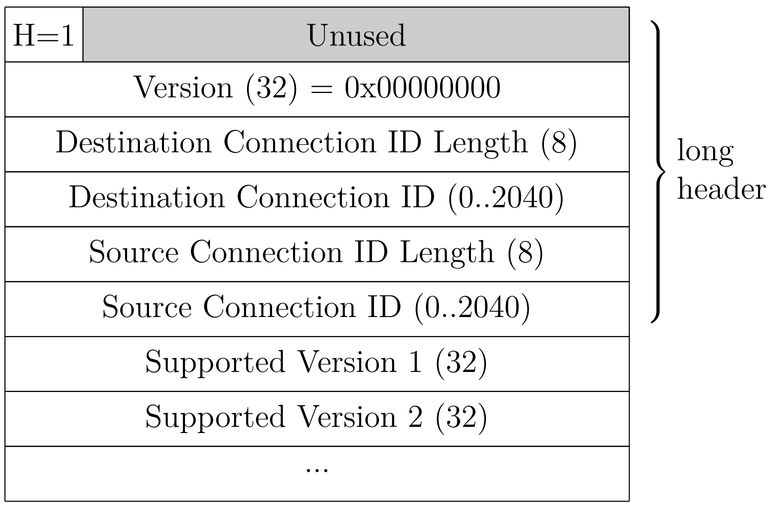
Source Connection ID
The Connection ID issued by the sender of the packet.

Reserved Bits
Bits reserved for use in the future QUIC versions. In the initial QUIC version, these bits must be set to 0.

2.2.2 Version Negotiation Packet

The Version Negotiation packet is sent by the server when it receives a long header packet requesting a version of QUIC which is not supported by the server. As an exception to other long header packets, the Version Negotiation packet is not discriminated by a specific value in the Packet Type field in the header, but by a special value 0x00000000 in the Version field.

After the header, the packet contains a list of supported versions, each one listed as a 32-bit integer in big-endian. Also, since future QUIC versions may allow larger Connection IDs than 20 bytes, a valid Version Negotiation packet can contain up to 255-byte Connection IDs. The structure of the Version Negotiation packet is illustrated in Figure 2.2.


H=1
Unused
Version (32) = 0x00000000
Destination Connection ID Length (8)
Destination Connection ID (0..2040)
Source Connection ID Length (8)
Source Connection ID (0..2040)
Supported Version 1 (32)
Supported Version 2 (32)
...
long header

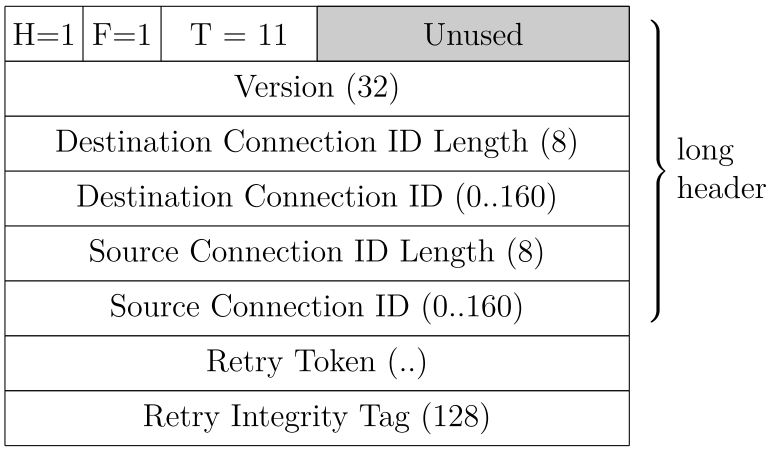
H = Header Form

Figure 2.2: Version Negotiation packet structure

The identifier of the initial QUIC version is 0x00000001. There are also unique identifiers for draft versions of the protocol valid only until the QUIC specification is finalized.

2.2.3 Retry Packet

Retry packets are distinguished by bits 11 in the *Packet Type* field of the long header. A server sends Retry packets as part of the optional *Address Validation* mechanism used to protect against traffic amplification attack described in section 2.8.3. Figure 2.3 illustrates the structure of the Retry packet.


H=1
F=1
T = 11
Unused
Version (32)
Destination Connection ID Length (8)
Destination Connection ID (0..160)
Source Connection ID Length (8)
Source Connection ID (0..160)
Retry Token (..)
Retry Integrity Tag (128)
long header

H = Header Form, F = Fixed Bit, T = Packet Type

Figure 2.3: Retry packet structure

The semantics of the fields specific to the Retry packet are:

Retry Token
Contains an opaque token generated by the server. This token must be

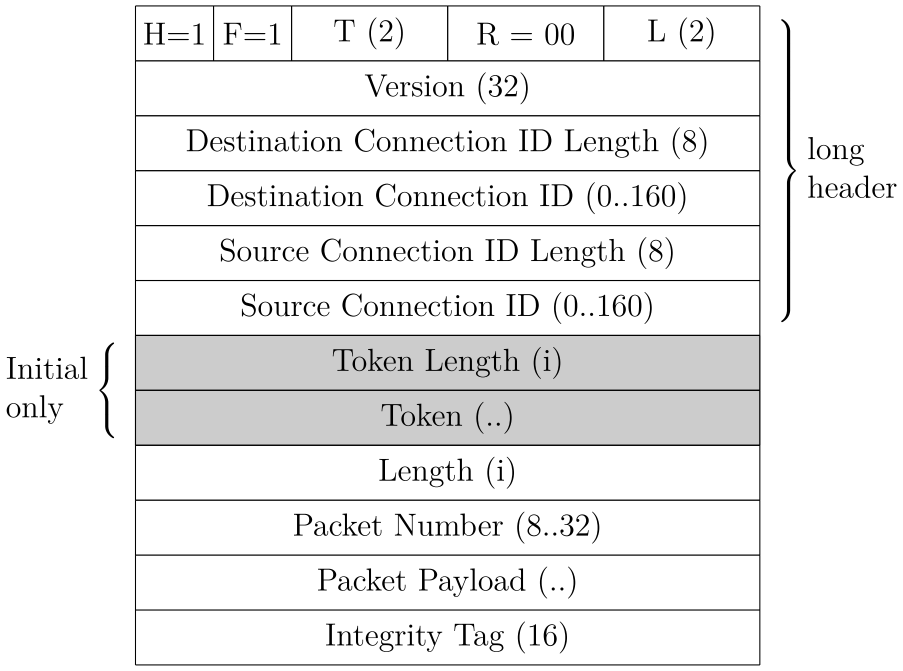
echoed back in an Initial packet during the next connection attempt. The token must be difficult to guess by the attacker and verifiable in a stateless manner, i.e., without saving it in memory for future comparisons.

Retry Integrity Tag

Tag used to check the integrity of the packet. Detailed information on how the integrity tag is calculated can be found in the TLS integration specification [19, Section 5.8].

2.2.4 Initial, Handshake, and 0-RTT Packets

Initial, Handshake, and 0-RTT packets are almost identical in structure. All three types use the *Type-Specific Bits* from the long header to store *Reserved Bits* and the *Packet Number Length.* After the long packet header, these packets contain the *Length* field containing the packet length, *Packet Number* field, and the actual payload consisting of QUIC frames. The only exception to this structure is the Initial packet, which contains two additional fields just after the long header: *Token Length* and *Token*, which are used to carry the Retry Token from the Retry packet in case Address Validation is requested by the server. Figure 2.4 illustrates the structure of these three packet types.



H = Header Form, F = Fixed Bit, T = Packet Type,
R = Reserved Bits, L = Packet Number Length

Figure 2.4: Structure of the Initial, Handshake, and 0-RTT packets

The semantics of the new fields in these packets are:

Token Length *(Initial only)*

Length of the Token field.

Token *(Initial only)*
Contains an opaque token if the server provided one in the Retry packet as part of Address Validation.

Reserved Bits (R)
Bits reserved for use in the future QUIC versions. In the initial QUIC version, these bits must be set to 0.

Packet Number Length (L)
The length of the encoding used for the packet number.

Length
The length of the remainder of the packet. This includes the Packet Number, the packet's payload, and the AEAD integrity tag.

Packet Number
The sequence number of this packet in the respective packet number space. This field uses the packet number encoding described in section 2.1.3.

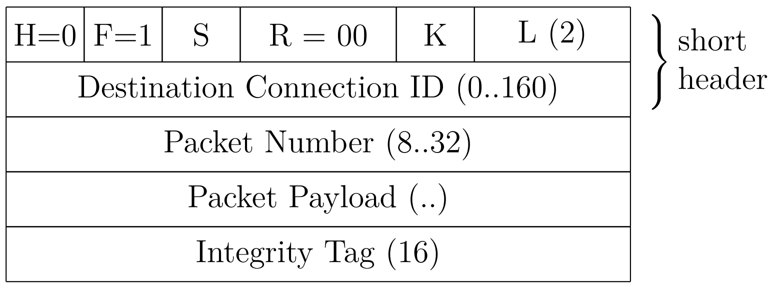
Packet Payload
Serialized sequence of QUIC frames.

Integrity Tag
Opaque checksum produced by the AEAD cipher during packet encryption. Packet encryption is described in detail in section 2.8.2.

2.2.5 1-RTT Packet

1-RTT packets are the only packet that uses the short header to make more space for application data in the UDP datagram. In addition to the Header Form, Fixed, Reserved Bits, and Packet Number Length fields, which have the same meaning as in the long header, the short header contains *Spin Bit* and *Key Phase Bit* fields.

1-RTT packets can be sent only after the connection has been successfully established. This implies that Connection IDs used for the connection by both endpoints are already known, and there is no need to repeat the SCID or specify the length of the DCID. 1-RTT packets also lack the Length field. It is assumed that 1-RTT packets fill the rest of the UDP datagram. Therefore, after the short header, only the Packet Number field and the QUIC frame payload follows. The structure of the 1-RTT frame is illustrated in Figure 2.5.



H = Header Form, F = Fixed Bit, S = Spin Bit,
R = Reserved Bits, K = Key Phase Bit, L = Packet Number Length

Figure 2.5: 1-RTT packet structure

The semantics of the fields specific to 1-RTT packets are:

Spin Bit
: A bit used for an optional QUIC feature which allows on-path nodes to measure connection latency by observing changes in this bit. For a detailed description of the Spin Bit feature, see the complete specification [18, Section 17.3.1].

Key Phase Bit
: A bit used to communicate that the keys used for the packet encryption need to be updated. After the keys have been used to encrypt a certain number of packets, it is necessary to update them in order to maintain the level of protection in the connection. section 2.8.2 describes the mechanism of Key Update in detail.

2.3 QUIC Frames

QUIC frames are low-level QUIC protocol messages carried in the payload of Initial, Handshake, 0-RTT, and 1-RTT packets. Examples of these frames include, e.g., `ACK` frames carrying acknowledgments for received packets, `STREAM` frames carrying the application data, and `CRYPTO` frames carrying data for the TLS handshake.

During the lifetime of the connection, all QUIC packets must be acknowledged by sending an ACK frame in another packet in the same packet number space. However, not all packets have to be acknowledged immediately; e.g., acknowledging packets containing only ACK would cause an endless flood of ACK packets. In such cases, sending an acknowledgement is delayed and sent later together with more urgent data. Frame types that require immediate acknowledgments are called *ack-eliciting frames*, and the packets with at least one such frame are called *ack-eliciting packets*.

Because packets in different packet number spaces offer a different level of confidentiality, not all frames can be sent in any packet type. For example, `STREAM` frames — which carry the application data — must not be sent in Initial and Handshake packets to avoid compromising security. Table 2.3 lists all frame types used in QUIC, whether they are ack-eliciting and which packets can carry them.

Table 2.3: QUIC frame types

		Allowed in packet type			
Frame type	Ack-eliciting	Initial	Handshake	0-RTT	1-RTT
`PADDING`		✓	✓	✓	✓
`PING`	✓	✓	✓	✓	✓
`ACK`		✓	✓		✓
`RESET_STREAM`	✓			✓	✓
`STOP_SENDING`	✓			✓	✓
`CRYPTO`	✓	✓	✓		✓
`NEW_TOKEN`	✓				✓
`STREAM`	✓			✓	✓
`MAX_DATA`	✓			✓	✓
`MAX_STREAM_DATA`	✓			✓	✓
`MAX_STREAMS`	✓			✓	✓
`DATA_BLOCKED`	✓			✓	✓
`STREAM_DATA_BLOCKED`	✓			✓	✓
`STREAMS_BLOCKED`	✓			✓	✓
`NEW_CONNECTION_ID`	✓			✓	✓
`RETIRE_CONNECTION_ID`	✓			✓	✓
`PATH_CHALLENGE`	✓			✓	✓
`PATH_RESPONSE`	✓			✓	✓
`CONNECTION_CLOSE`		✓	✓	✓	✓
`HANDSHAKE_DONE`	✓				✓

2.4 QUIC Connection

This section describes the details of Connection ID management and the QUIC connection lifetime.

2.4.1 Connection ID

Traditional network protocols use the combination of remote endpoint IP address and port to identify the connection. QUIC, on the other hand, uses a dedicated Connection ID identifier. This, in essence, enables migrating the connection to different network paths and interfaces, e.g., from cellular data to a Wi-Fi network, because the connection identity does not depend on the peer's IP address.

Connection IDs are opaque byte sequences between 8 to 20 bytes in length. Each endpoint in a QUIC connection independently selects Connection IDs it will use to identify the QUIC connection. These Connection IDs are then used to populate the Source Connection ID and Destination Connection ID fields of the QUIC packets.

The endpoints exchange the first pair of Connection IDs during connection establishment. Additional Connection IDs can be issued independently by each endpoint during the connection's lifetime using the `NEW_CONNECTION_ID` frame. These additional Connection IDs are primarily used when migrating the connection to a new network path because QUIC specification requires that the same

Connection ID be used only on one network path in order to prevent correlation of the network traffic by external observers.

Connection IDs can also be retired by the other endpoint using the `RETIRE_-CONNECTION_ID` frame. By retiring a Connection ID, the endpoint communicates that it will no longer use the Connection ID to send packets and that the other endpoint should drop any incoming packets that use that Connection ID. At the same time, retiring a Connection ID also serves as a request to the peer to issue a new Connection ID as a replacement.

Alternatively, endpoints can use a zero-length Connection ID. In that case, the connection's identity is tied to the remote endpoint's IP address and port. Using zero-length Connection ID saves space in the sent datagrams but imposes several limitations on the connection. For example, if an endpoint uses a zero-length Connection ID, it cannot issue additional connection IDs and, therefore, it cannot migrate a connection to a new local address.

2.4.2 Matching Packets to Connections

When a packet arrives at an endpoint, it needs to be associated with an existing connection or — for servers — potentially initiate a new connection. If packet contains a non-zero-length Connection ID in the DCID field, the Connection ID is used to find an existing connection. If the packet uses a zero-length Connection ID, the local and remote addresses determine the target connection.

Figure 2.6 illustrates how server endpoint processes incoming packets from client connections. The server maintains a table mapping between DCIDs corresponding connections and dispatches the incoming packets accordingly. If a packet cannot be associated with an existing connection, it may be a new connection attempt; otherwise, the packet is discarded.

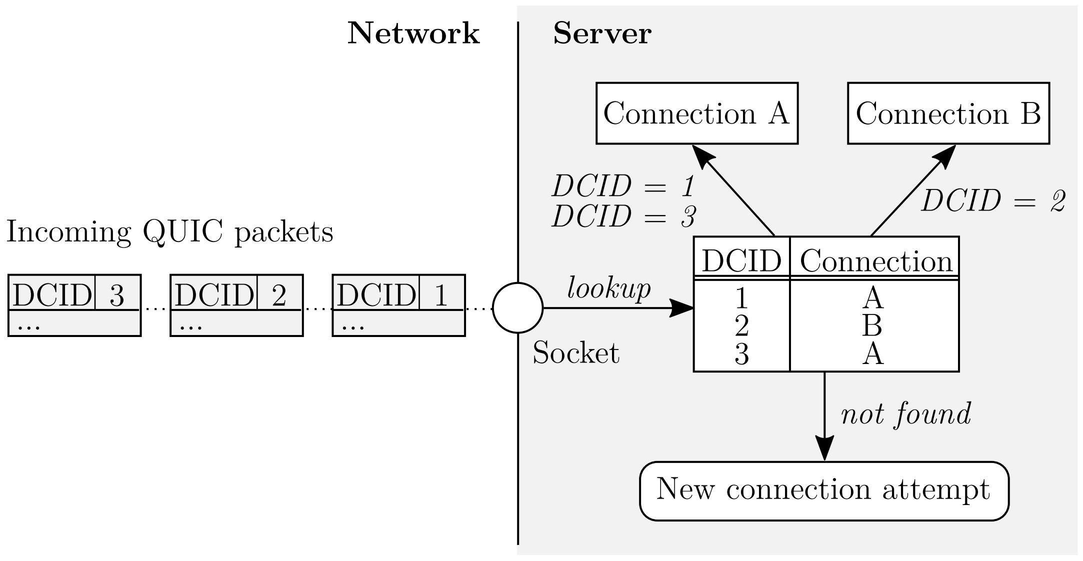


Figure 2.6: Multiple QUIC connections on the same machine port

In case the packet cannot be associated with an existing connection, client endpoints simply ignore the packet. Server, on the other hand, check the packet type and version of the protocol it requires. For valid Initial packets with supported versions, the server proceeds with the handshake described in the previous section. For invalid Initial packets, the server responds with an Initial packet containing a `CONNECTION_CLOSE` frame signaling the refusal of the connection.

In case the client packet requests an unsupported QUIC version, the server replies with a Version Negotiation packet (described in section 2.2.2). After receiving such a packet, the client can try again using one of the supported versions. Figure 2.7 illustrates such an exchange. In the figure, the client tries to establish a connection using an unsupported version denoted by X. The server then replies with a Version Negotiation packet listing versions 1, 2, and 3. The client then tries to establish the connection again using version 1.

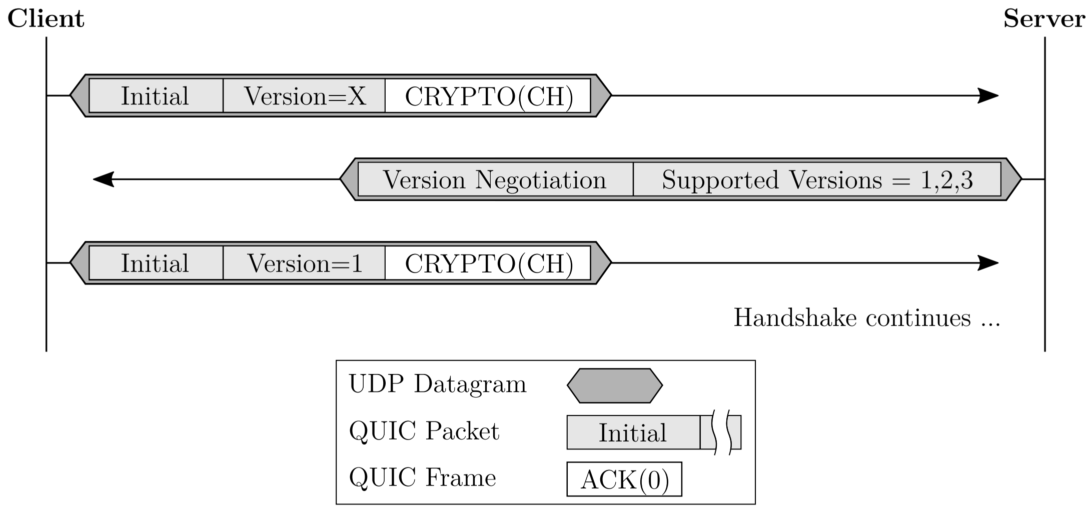


Figure 2.7: Example Version Negotiation packet exchange

Lastly, the server may choose to send a Stateless Reset (see later in section 2.4.5) for any other packet that it cannot match to an existing connection.

2.4.3 QUIC Transport Parameters

QUIC has several options for parameterizing the connection. These are called *Transport Parameters*, and they essentially represent constraints for the other endpoint. Each endpoint sets the transport parameters for the other endpoint. QUIC leverages the extensibility of the TLS protocol to exchange transport parameters during the connection handshake. Transport parameters contain information such as

- initial flow control limits,
- whether connection migration is allowed,
- maximum delay before sending an acknowledgment for ack-eliciting packets,
- maximum idle timeout before the connection is silently closed, and
- maximum size of a UDP packet the endpoint is willing to receive.

Many parameters have a default value, which is used when the given transport parameter is not sent. Other transport parameters are mandatory. The exhaustive list of the transport parameters can be found in the core transport specification [18, Section 7.3].

2.4.4 Connection Establishment

To initiate a new connection, clients send an Initial packet to the server, which initiates the handshake process. After the handshake completes, both peers have derived protection keys necessary to send and receive 1-RTT packets with application data.

Figure 2.8 illustrates a possible sequence of QUIC packets sent during a connection handshake. The figure lists the contents of the individual QUIC frames, including the TLS records sent inside `CRYPTO` frames. However, contents of the `CRYPTO` frames are listed only for illustrative purposes because QUIC itself does not interpret their contents in any way. Instead, the contents of `CRYPTO` frames are forwarded to a TLS implementation.

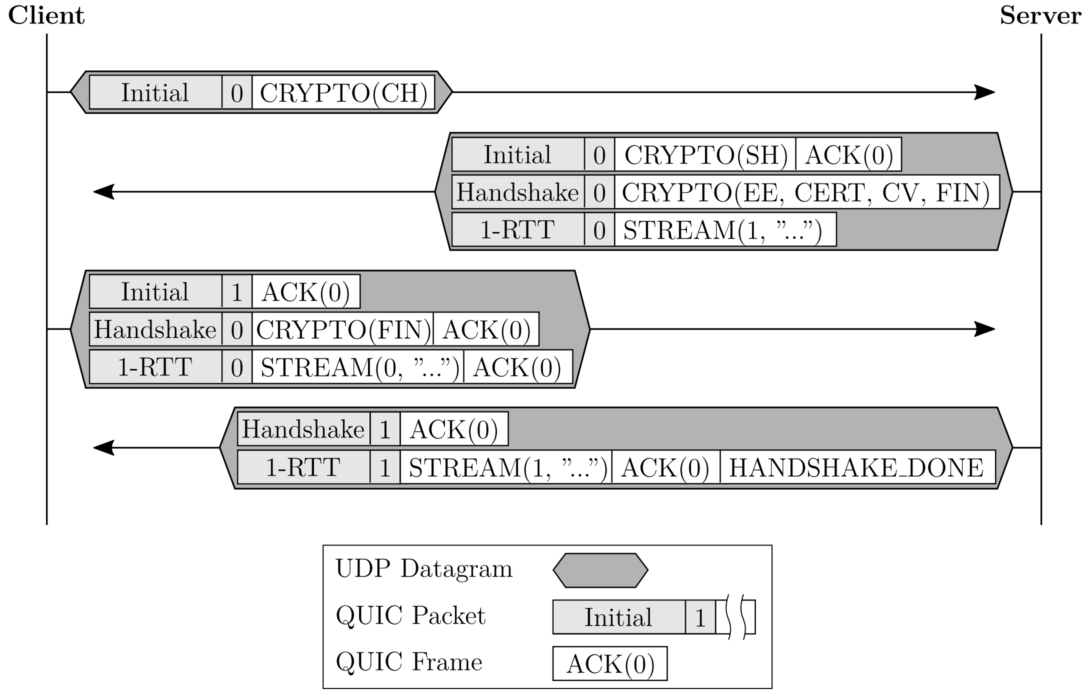

Client
Server
Initial | 0 | CRYPTO(CH)
Initial | 0 | CRYPTO(SH) | ACK(0)
Handshake | 0 | CRYPTO(EE, CERT, CV, FIN)
1-RTT | 0 | STREAM(1, "...")
Initial | 1 | ACK(0)
Handshake | 0 | CRYPTO(FIN) | ACK(0)
1-RTT | 0 | STREAM(0, "...") | ACK(0)
Handshake | 1 | ACK(0)
1-RTT | 1 | STREAM(1, "...") | ACK(0) | HANDSHAKE_DONE
UDP Datagram
QUIC Packet
Initial | 1
QUIC Frame
ACK(0)

Figure 2.8: Example QUIC handshake flow

In its first datagram, the client sends an Initial packet with initial information consisting of a single `CRYPTO`(CH) frame. This frame contains the *Client Hello* TLS message.

The server replies with a UDP datagram containing three coalesced QUIC packets. The first is an Initial packet that acknowledges the client's Initial packet using the `ACK`(0) frame, and a `CRYPTO`(SH) frame with a *Server Hello* message. The TLS implementation uses the contents of Client Hello and Server Hello messages to derive the Handshake protection keys. The server then advances the TLS handshake by sending another `CRYPTO` frame in the Handshake packet. The server also has enough information to derive the 1-RTT keys, so it can also start sending data on Stream 1 using the `STREAM`(1, "...") frame in a 1-RTT packet.

Because the server's Initial packet contained an ack-eliciting `CRYPTO`(SH) frame, the client needs to acknowledge it by sending an Initial frame with an `ACK`(0) frame. The client possesses both Client and Server Hello messages and can derive the Handshake keys, enabling him to process the server's Handshake packet. The Handshake packet needs to be separately acknowledged by another `ACK`(0) frame, and a reply from the TLS layer must be sent using another `CRYPTO` frame. From the information in the server's Handshake packet's `CRYPTO` frame, the client derives 1-RTT protection keys and processes the server's 1-RTT packet. In addition to sending an `ACK`(0) frame for the server's 1-RTT packet, the client can now start sending application data on stream 0 using a `STREAM`(0, "...") frame.

After the server detects that the handshake has completed, it sends a `HANDSHAKE_DONE` frame to communicate it to the client. The figure also shows that servers sends `ACK`(0) frame for the clients Handshake packet but not for the client's last Initial packet because it was not ack-eliciting.

2.4.5 Connection Termination

QUIC connection can be terminated in three ways:

- Idle timeout
- Immediate close
- Stateless reset

Idle Timeout

If idle timeout is enabled, the endpoint silently closes the connection if it does not receive a packet from the peer for a specified time period. Each peer may advertise a timeout period using the `max_idle_timeout` transport parameter, but the effective value is the minimum of the two values.

In order to prevent timeouts, endpoints can send a `PING` or another ack-eliciting frame to test the liveness of the connection. However, sending `PING` frames should be initiated by the application protocol, not QUIC implementation, to prevent unnecessary network traffic.

Immediate Close

An immediate close can be initiated either by QUIC implementation or by the application. Either of the endpoints can initiate an immediate close by sending a `CONNECTION_CLOSE` frame. By sending a `CONNECTION_CLOSE` frame, the peer enters a *closing state*, in which it includes the `CONNECTION_CLOSE` frame in all packets, it sends in reply to incoming packets. The closing state is also entered if the endpoint receives a `CONNECTION_CLOSE` frame from the peer. In that case, the endpoint also echoes the `CONNECTION_CLOSE` frame back to the other endpoint.

The closing state lasts until the endpoint is sure the other endpoint is also in the closing state — e.g., until it also receives `CONNECTION_CLOSE` — or until a *closing timeout* expires. The closing timeout period is calculated from the current estimate of the round-trip time of the connection.

The `CONNECTION_CLOSE` frame carries an error code and, optionally, a human-readable error phrase. When initiated by QUIC, the error codes semantics are defined by the QUIC specification. However, when initiated by the application protocol, the semantics of all possible error code values sent in the frame are defined by the application protocol itself. This implies that the implementations must require an error code when closing the connection and should not provide a default error code value.

Stateless Reset

A stateless reset is an option of last resort for an endpoint that does not have access to the state of a connection, possibly as a result of a crash or outage. An endpoint may send a stateless reset in response to receiving a packet that it cannot associate with an active connection.

In such cases, the endpoint sends a specially crafted packet that ends with a Stateless Reset Token generated for the DCID from the incoming packet. The Stateless Reset Token requirements are quite complex, and we encourage readers to read the full specification if they are interested in details [18, Section 10.4].

2.4.6 Connection Migration

A novel feature of QUIC is migrating connections to a different network path. This is enabled by using a dedicated Connection ID instead of using the endpoint's address to identify the connection. In the initial QUIC version, only client endpoints can migrate the connection to a different address.

Before migrating a connection, the endpoint can optionally check the reachability of the other endpoint using the process called *path validation*. Path validation consists of exchanging *probing packets*, and is described in section 2.8.4. The migration itself is initiated by client by simply switching to the new local address for sending all outbound packets. After the server receives the first non-probing packet from the new client's address, it sends all future packets to that new address.

QUIC uses additional measures to prevent network traffic from being correlated by the outside observers. In section 2.4.1 we mentioned that each endpoint could use multiple Connection IDs to refer to the same connection. QUIC requires that each Connection ID be used only on one network path. Therefore, both endpoints must switch to using different Connection IDs when the connection is migrated. Therefore, connection migration can only be initiated if both endpoints issued additional Connection IDs using the `NEW_CONNECTION_ID` frame.

The connection migration process is illustrated in Figure 2.9. Client and servers use Connection IDs **C1** and **S1**, respectively. The client first probes the server's reachability with a probing packet containing a `PATH_CHALLENGE` frame. Because this packet is sent from a different local address, it uses a different Connection ID (**S2**) issued previously by the server. Likewise, the server uses a different Connection ID **C2** to reply with a packet containing a `PATH_RESPONSE` frame, confirming the reachability from the client's new local address. The server, meanwhile, still uses the old network path for all other communication with the client. After confirming reachability, the client migrates the connection by sending all packets via the new local address. After receiving the next non-probing packet from the new address, the server switches to the new client's address for all outgoing packets. From that point, Connection IDs **C1** and **S1** are no longer used.

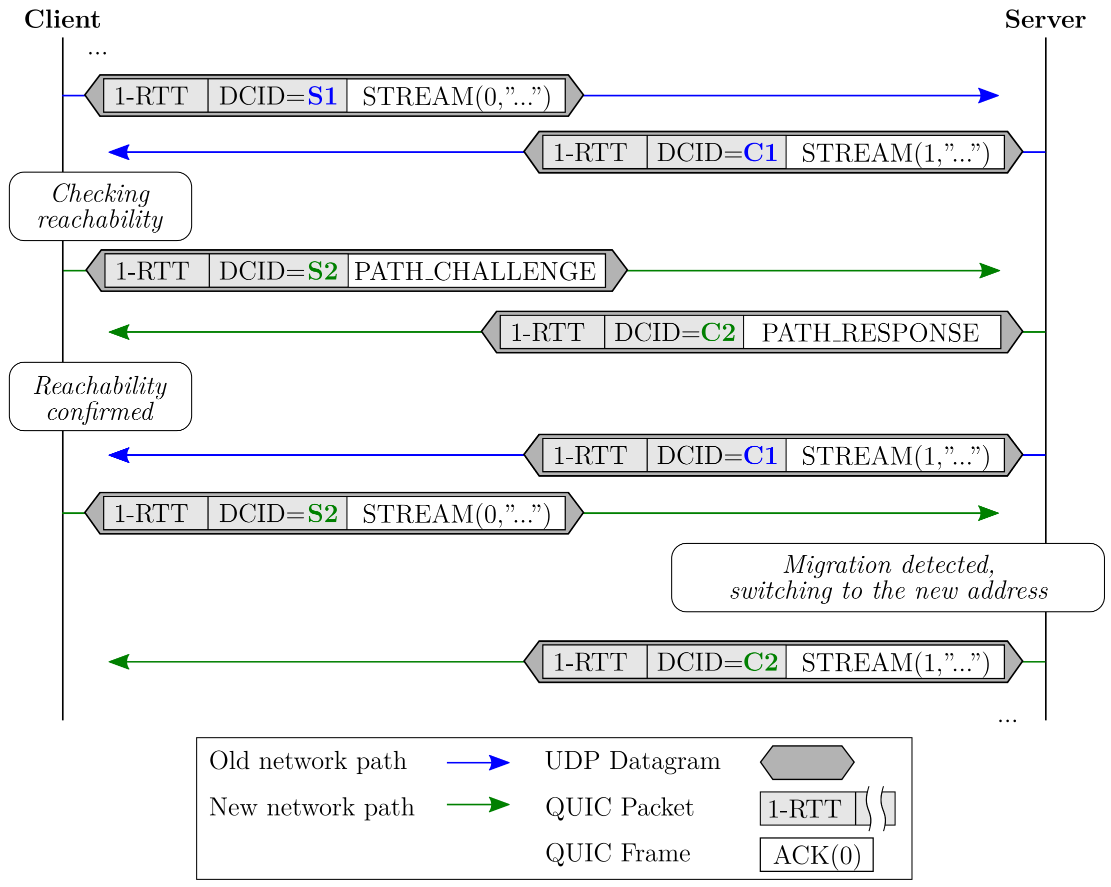


Figure 2.9: Flow of packets during connection migration

Connection migration is an optional feature and can be disabled by sending the `disable_active_migration` transport parameter during the connection handshake.

2.5 QUIC Streams

QUIC can transport multiple streams of data in a single connection. Each stream is identified by its *Stream ID* and is processed independently of the other streams. Each QUIC packet can carry data for one or more QUIC streams. Figure 2.10 illustrates how QUIC may pack two streams into QUIC packets such that those streams are transported in parallel.

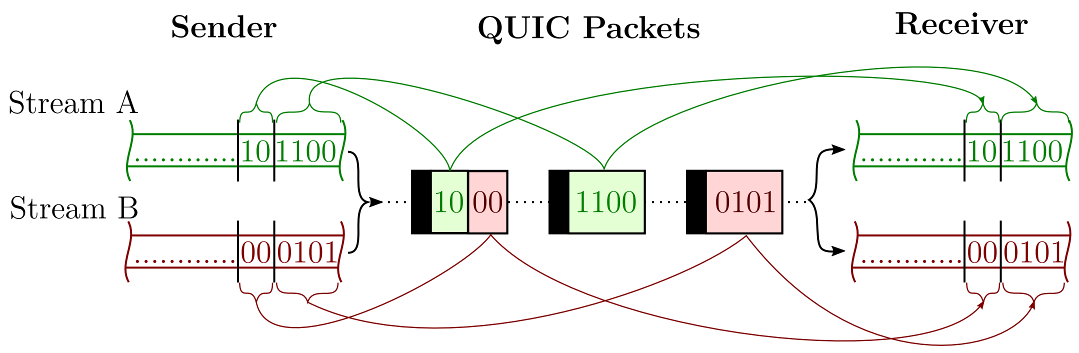


Figure 2.10: Stream multiplexing in QUIC

2.5.1 Streams Types

Streams transported by QUIC can be either unidirectional or bidirectional. Unidirectional streams carry data from the initiator to its peer, and bidirectional streams carry data in both directions. Both client and server can open new streams. QUIC recognizes four types of streams, and the type of the stream is encoded in the two least significant bits of the Stream ID. The stream types and their associated encoding is summarized in Table 2.4.

Table 2.4: Mapping of QUIC Stream types to Stream ID bits

Stream type	Least significant bits
Client-Initiated, Bidirectional	00
Server-Initiated, Bidirectional	01
Client-Initiated, Unidirectional	10
Server-Initiated, Unidirectional	11

Bidirectional streams can be viewed as the combination of two unidirectional streams in opposing directions. After opening a bidirectional stream, each direction of the stream behaves as separate inbound and outbound unidirectional streams. This implies, e.g., that the sending and receiving parts of the bidirectional stream can be closed independently of each other.

2.5.2 Stream Lifetime

Opening a stream does not require any particular action. Streams are opened simply by sending the first `STREAM` frame carrying data for that stream. However, streams of a particular type can be opened only in ascending order of their Stream IDs. For example, a stream 2 must be opened before opening stream 6. Sending data for higher-numbered streams will automatically open all lower-numbered streams of the same stream type.

Streams can be closed either gracefully or abortively. Graceful stream close is signaled by a *Fin* bit in the `STREAM` frame, signaling that data carried by this packet are the last part of the stream. The stream then is gracefully closed once all stream data is confirmed received by the other endpoint.

Abortive stream close is achieved using the `RESET_STREAM` frame and, therefore, this action is also referred to as *resetting the stream.* Streams can be reset only by the sender. The receiver can request aborting the stream by sending a `STOP_SENDING` frame which indicates that the application no longer wishes to receive data from that particular stream. Both `RESET_STREAM` and `STOP_SENDING` frames carry an application-level error code which is reported to the applicaiton on the other side.

After the stream is closed, its Stream ID may not be reused. Instead, the next available Stream ID must be used. QUIC uses the variable-length integer encoding (see section 2.1.3) and, therefore, there is no shortage of available stream IDs, which range from 0 to $2^{62} - 1$.

2.5.3 Required Operations on Streams

The implementation should provide the following operations on sending part of the stream:

- write data;
- end the stream by specifying that all data has been written; and
- terminate the stream with an application-level error code.

On receiving part of the stream, application protocols must be able to:

- read data; and
- abort reading with an application-level error code.

2.6 Flow Control

QUIC aims to be a general-purpose transport protocol to be used over a potentially untrusted network, and as such, it needs to protect endpoints from malicious peers. To prevent malicious senders from exhausting all available memory on the receiver by sending large amounts of data, or fast senders from overwhelming slow receivers, QUIC employs a credit-based flow control scheme.

All QUIC streams are flow controlled both individually and together as an aggregate. Each endpoint also controls the number of streams the other peer is allowed to open. All flow control limits are communicated to the peer using three types of frames:

- `MAX_STREAM_DATA`: maximum offset of data sent on a stream with specified Stream ID.
- `MAX_DATA`: the maximum sum of all offsets of data sent on all streams.
- `MAX_STREAMS`: the maximum number of streams of a particular stream type.

Endpoints can only increase the flow control limits. Their peers must ignore any attempts to decrease the flow control limits to ensure consistency when two consecutive QUIC packets with flow control updates are reordered during transit. In case the peer violates any of the control flow limits mentioned above, the QUIC implementation must immediately terminate the connection.

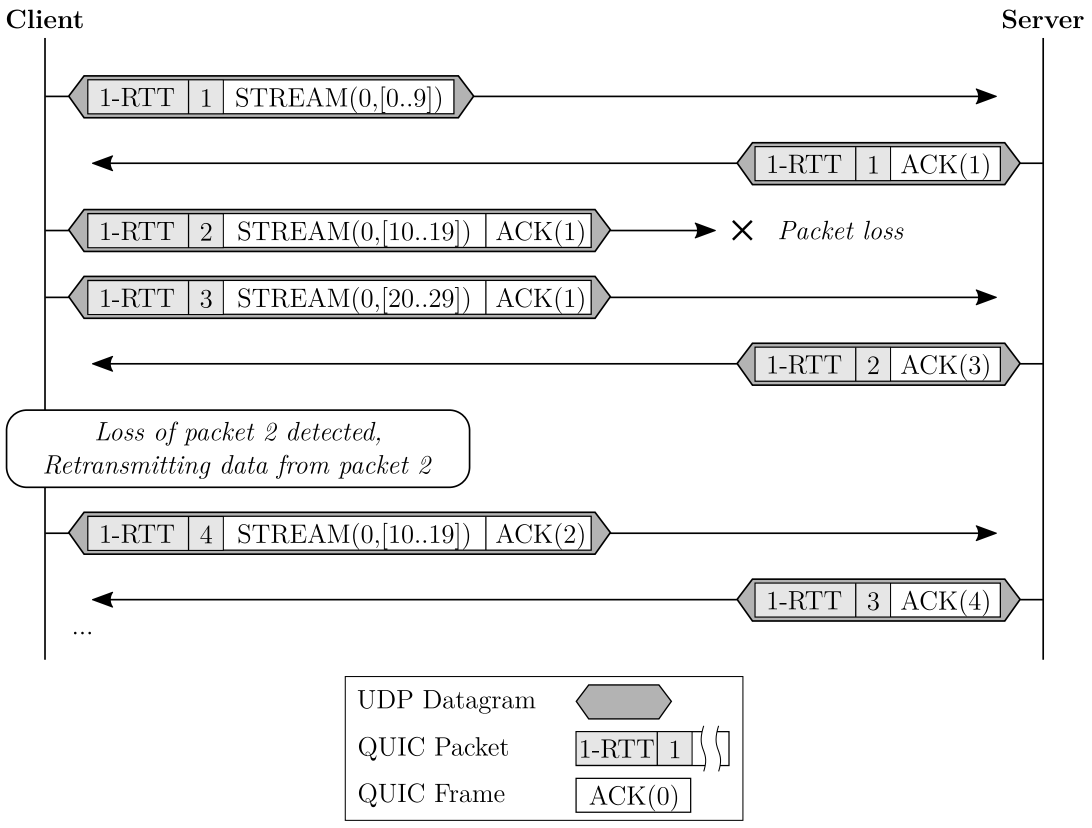
2.7 Loss Detection and Recovery

Because UDP is an unreliable transport protocol, QUIC must implement measures to recover from packet loss. The packet loss detection is implemented similarly to TPC — endpoints send acknowledgments for each received packet. However, an essential difference from TCP is that QUIC endpoints do not retransmit entire lost packets with the same packet number. Instead, each QUIC frame in the original packet is updated and sent in some future packet or dropped altogether if the information contained in the frame is no longer relevant.

The endpoints communicate the acknowledgment using the `ACK` frame containing ranges of received packet numbers. A packet number can be sent multiple times (in `ACK` frames in different packets) until the endpoint can determine that the other endpoint received the acknowledgment.

For packets containing an *ack-eliciting* frame (see section 2.3), acknowledgements must be sent within the period specified by the `max_ack_delay` transport parameter. For other packets, such as packets containing only an `ACK` frame, the acknowledgement can be delayed until an ack-eliciting packet is received.

Figure 2.11 illustrates the loss detection and retransmission process in action. In the figure, frame `STREAM`(0,[0..9]) denotes a `STREAM` frame carrying the first 10 bytes of the stream with Stream ID 0. When the server receives the acknowledgment for packet 3, but not for packet 2 sent earlier, it infers that packet 2 never reached the receiver and retransmits the bytes 10 to 19 of stream 0 in packet 4. The server does not have to retransmit the `ACK`(1) frame because it was sent in packet 3, which the client acknowledged. Therefore, packet 4 acknowledges only the client's packet 2.



Client
Server
1-RTT 1 STREAM(0,[0..9])
1-RTT 1 ACK(1)
1-RTT 2 STREAM(0,[10..19]) ACK(1)
Packet loss
1-RTT 3 STREAM(0,[20..29]) ACK(1)
1-RTT 2 ACK(3)
Loss of packet 2 detected,
Retransmitting data from packet 2
1-RTT 4 STREAM(0,[10..19]) ACK(2)
1-RTT 3 ACK(4)
...
UDP Datagram
QUIC Packet
1-RTT 1
QUIC Frame
ACK(0)

Figure 2.11: Loss detection and retransmission example

The exact criteria for a packet to be deemed lost by a QUIC endpoint are following:

1. The packet was not acknowledged.

2. A packet which was sent later has been acknowledged.

3. Either the packet has been sent long enough in the past, or its packet number is sufficiently smaller than the highest acknowledged packet number.

The third condition is dependent on the values of particular constants. The specification recommends that the packet is considered lost if the gap between its packet number and the highest acknowledged packet number is at least 3, or if it was sent for longer than 9/8 times the estimate of the current round-trip time.

The first condition mentioned above requires receiving a packet from the peer to declare any packet as lost. However, the loss of the last packet in a sequence could go undetected because there is no following packet that can be acknowledged. In order to avoid possible deadlocks in such scenarios, QUIC endpoint sends up to two ack-eliciting *probe packets* if it does not receive a packet from a peer in a period called *probe timeout* (PTO for short). The PTO duration doubles each time probe packets are sent until either a reply is received or the connection is terminated due to idle timeout (see section 2.4.5)

Similarly to TCP, QUIC also uses congestion control to manage the *congestion window* — the amount of data that can be in-flight. The selection of the congestion control algorithm is left on the implementation. As an example, The QUIC specification document for loss detection and recovery [20, Section 7] describes a congestion control algorithm similar to TCP NewReno [21] algorithm.

2.8 Security

This section describes the mechanisms used to ensure the security of the protocol. Besides encrypting all packets sent throughout the lifetime of the connection, QUIC uses additional mechanisms to ensure that the servers using the protocol are resistant to denial-of-service and other cyber-attacks.

2.8.1 TLS Integration

Instead of designing a new handshake protocol, QUIC offloads the encryption negotiation to TLS protocol (more precisely, TLS version 1.3). The low-level messages used in TLS, such as *Server Hello* and *Client Hello* are transported by QUIC inside `CRYPTO` frames (as illustrated in section 2.4.4) and passed to a TLS implementation on the other side. This way, QUIC can offer the same confidentiality level as conventional TLS connections.

The TLS protocol is extensible. Among the standard extensions which are also used by QUIC are Application-Layer Protocol Negotiation (ALPN) [22] and Server Name Indication (SNI) [23].

ALPN is used when multiple application protocols or their multiple versions are supported on the same TCP or UDP port. ALPN allows the application layer to negotiate — as part of the TLS handshake — which application protocol will be used in the established connection.

Clients use SNI to specify the hostname of the server to which they are connecting. When multiple websites are hosted on the same IP address and port, SNI allows the server to customize the security configuration for each hosted website. During connection establishment, proper security configuration, such as the SSL certificate to be used, can be selected based on the hostname provided by the client.

QUIC also uses a custom TLS extension to exchange transport parameters during the handshake. The QUIC transport parameters were described in section 2.4.3.

2.8.2 Packet Protection

All QUIC packets of type Initial, Handshake, 1-RTT, and 0-RTT are encrypted to ensure the integrity and confidentiality of the transmitted data. Negotiation of the cryptographic ciphers and the encryption keys is handled by the TLS handshake. This section focuses on how the negotiated encryption is applied to QUIC packets.

Authenticated Encryption with Associated Data

QUIC uses type of encryption called Authenticated encryption with associated data (AEAD) [24]. This type of encryption ensures both the confidentiality and authenticity of the encrypted data. In addition to encrypted data — called *ciphertext* — AEAD encryption outputs an authentication tag, which is used to check the integrity of the payload during decryption. The encryption can be authenticated by supplying additional authentication data (AAD for short), which are not encrypted but influence the authentication tag and, therefore, must also be supplied during decryption. As additional protection, AEAD also accepts a *nonce* parameter, which is an additional input that is supposed to be unique for each encrypted message. By using a unique nonce parameter for each message, the algorithm ensures that two otherwise identical messages produce different ciphertexts.

The programming interface for AEAD provides the following operations:

- Encryption:
  - input: plaintext, key, nonce, AAD (optional)
  - output: ciphertext, authentication tag
- Decryption:
  - input: ciphertext, key, nonce, authentication tag, AAD (if provided during encryption)
  - output: plaintext or error if the authentication tag does not match the rest of the input

Deriving QUIC Protection Keys

QUIC derives multiple distinct keys that are used in different parts of the packet protection procedure. Furthermore, each packet type uses a different set of such keys. This allows QUIC to provide confidentiality and integrity protection even for Initial and Handshake packets, which are sent before the TLS handshake negotiates the application-level encryption.

Each set of encryption keys is derived from a *secret* — a sequence of bytes known only to the two endpoints. Secrets for deriving the Handshake and 1-RTT keys are produced by the TLS handshake. Secrets for Initial keys are derived using the DCID of the client's first Initial packet and a version-specific *initial salt* which ensures that newer versions of QUIC derive different protection keys. The derivation process uses the HKDF-Extract and HKDF-Expand-Label functions from RFC 5869 [25] to derive secrets of the desired length. The following equations show how QUIC derives distinct secrets for client and server.

$$\begin{aligned} \mathit{initial\_secret} &= \text{HKDF-Extract}(\mathit{dcid}, \mathit{initial\_salt}) \\ \mathit{client\_secret} &= \text{HKDF-Expand-Label}(\mathit{initial\_secret}, \texttt{"client in"}, \texttt{""}, 32) \\ \mathit{server\_secret} &= \text{HKDF-Expand-Label}(\mathit{initial\_secret}, \texttt{"server in"}, \texttt{""}, 32) \end{aligned}$$

Once the necessary secrets are obtained, a total of three keys are derived using the following equations.

$$\begin{aligned} \mathit{key} &= \text{HKDF-Expand-Label}(\mathit{secret}, \texttt{"quic key"}, \texttt{""}, 32) \\ \mathit{iv} &= \text{HKDF-Expand-Label}(\mathit{secret}, \texttt{"quic iv"}, \texttt{""}, 12) \\ \mathit{hp} &= \text{HKDF-Expand-Label}(\mathit{secret}, \texttt{"quic hp"}, \texttt{""}, 32) \end{aligned}$$

The following subsections describe how the *key*, *iv*, and *hp* keys are used in the actual process of packet encryption.

Packet Protection Procedure

When encrypting the packets, QUIC first encrypts the packet payload — the sequence of QUIC frames — using the AEAD cipher negotiated by the TLS implementation. QUIC specification allows the use of all AEAD ciphers allowed in TLS 1.3. These ciphers are:

- TLS_AES_128_GCM_SHA256
- TLS_AES_256_GCM_SHA384
- TLS_CHACHA20_POLY1305_SHA256
- TLS_AES_128_CCM_SHA256
- TLS_AES_128_CCM_8_SHA256

The parameters for AEAD for packet payload protection are:

- *key*: the *key* derived in the previous section
- *nonce*: the *iv* derived in the previous section, with the last 8 bytes XORed with the packet number
- *AAD*: the contents of the packet header
- *plaintext*: the payload of the packet.

The produced authentication tag is appended to the packet as the Integrity Tag field and, in case of the packet types with long headers, its size is included in the payload size in the Length field.

QUIC also protects the header of the packet. The header protection mechanism is more complicated than that of the payload protection. The process requires calculating a *header protection mask*, which is then applied using XOR to selected fields of the packet header and the Packet Number field. The algorithm for calculating the header protection mask depends on the negotiated cipher, but it always uses the *hp* key and a 16-byte sample of the encrypted payload.

The parts of the packets protected by the payload encryption and header protection mechanisms are illustrated in Figure 2.12 for long headers and in Figure 2.13 for short headers.

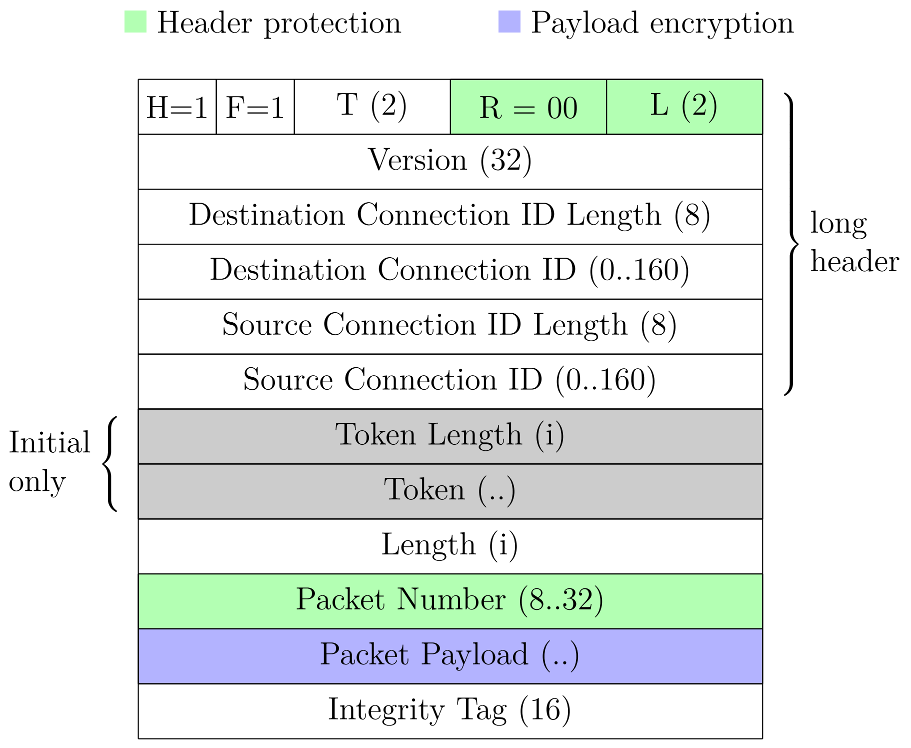


Figure 2.12: Protected fields in the long packet header

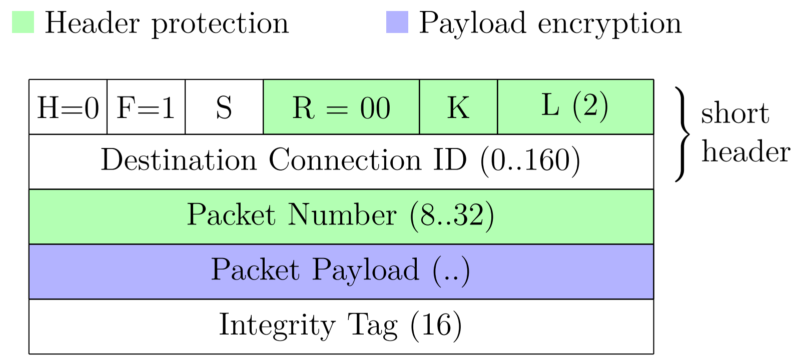


Figure 2.13: Protected fields in the short packet header

Updating 1-RTT Protection Keys

AEAD ciphers slowly lose the levels confidentiality and integrity protection with each encrypted packet. Therefore, QUIC — like TLS — tracks the number of packets encrypted using a particular encryption key and updates the key before an attacker can gain substantial advantage from observing the encrypted packets. Analysis in the specification document shows that updating the protection keys after sending at most 2^{23} packets provides the same confidentiality level as provided by TLS [19, Appendix B].

The protection keys are updated using a process called *key update*. The process is signalled to the peer by flipping the *Key Phase* bit in the 1-RTT packet header. After observing a change in the *Key Phase*, an endpoint derives new secret and *key* and *iv* keys using the HKDF-Expand-Label function:

$$\begin{aligned}
secret_{n+1} &= \text{HKDF-Expand-Label}(secret_n, \texttt{"quic ku"}, \texttt{""}, 32)\\
key_{n+1} &= \text{HKDF-Expand-Label}(secret_{n+1}, \texttt{"quic key"}, \texttt{""}, 32)\\
iv_{n+1} &= \text{HKDF-Expand-Label}(secret_{n+1}, \texttt{"quic iv"}, \texttt{""}, 12)
\end{aligned}$$

Because the *Key Phase* bit is protected by header protection, the *hp* key must remain unchanged to ensure that the other endpoint can correctly remove the header protection.

2.8.3 Client Address Validation

After receiving the first Initial packet from a new client, the server can request address validation by sending a Retry packet (see section 2.2.3). The Retry packet carries a token, which the client must echo back to the server in all following Initial packets. As long as an attacker cannot generate a valid token for its address, and the client can return that token, this exchange proves to the server that the client has received the token.

Figure 2.14 illustrates the use of Retry packet to validate client address during connection establishment. By default, clients do not fill the Token field of the Initial packet. The server rejects the initial connection attempt and issues a Retry Token (denoted “ABCD” in the figure). The client then tries again with another Initial with the provided token, and the server proceeds with the usual handshake.

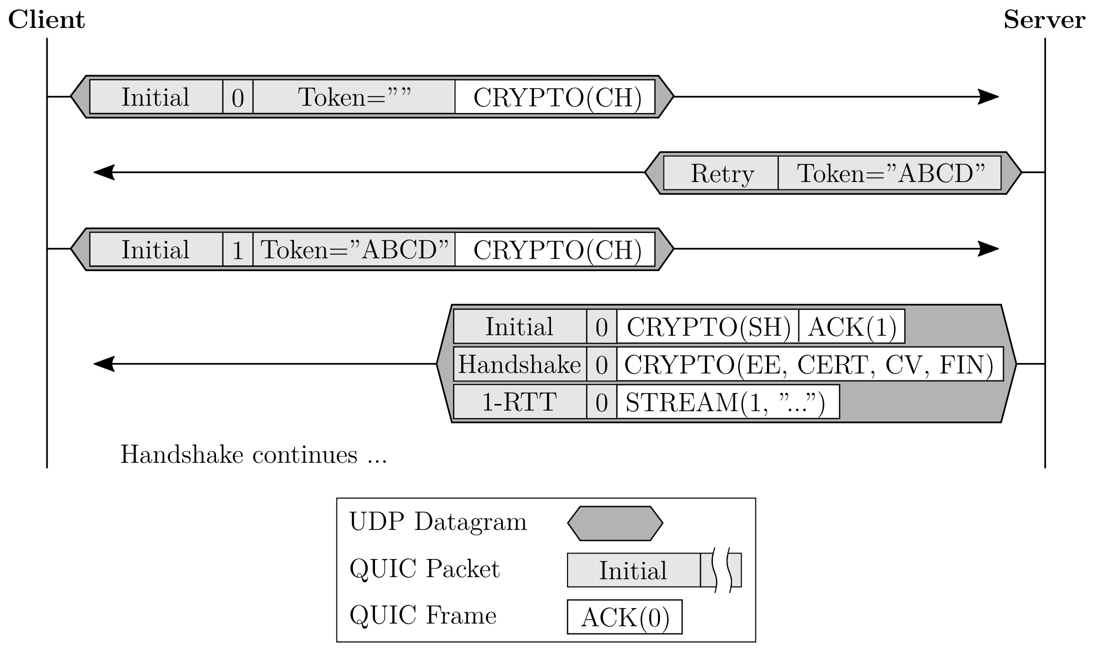


Figure 2.14: Client address validation using a Retry packet.

2.8.4 Path Validation

QUIC is layered on top of UDP, which is a connection-less protocol. This means that changes in endpoint address can also happen without active migration on the endpoint's part, e.g., because of NAT rebinding along the network path.

The other endpoint may also spoof the endpoint address in an attempt to perform *traffic amplification attack*. For this reason, the amount of data sent to the new endpoint address must be limited until *path validation* determines that the address belongs to the endpoint. Path validation is performed by sending a *probing packet* containing a `PATH_CHALLENGE` frame with an unpredictable token. The other endpoint must echo the token back in a `PATH_RESPONSE` frame. After that, the new address is considered validated, and the sending rate restrictions are lifted.

3. Analysis

This chapter analyzes the protocol and selects the necessary subset needed to evaluate a .NET implementation of the QUIC protocol. Afterward, we design the architecture and outline the implementation of its major parts. Lastly, we investigate the means for unit testing and debugging of the implementation.

3.1 Implemented Feature Subset Selection

The decision which features of the QUIC protocol we implement is guided by the goals we set in section 1.4. For feature selection, these goals can be rephrased into the following:

1. Support basic data transport, enabling some experimentation with QUIC as the transport for application layer protocols. *(goals 1 and 2)*
2. Enable performance measurements that are representative of the potential full QUIC implementation. *(goals 1 and 3)*

The first goal requires full implementation of the multiplexed stream abstraction as defined by the QUIC specification. It also requires implementing loss detection and recovery to ensure that no data gets lost during the transport.

In order to get representative performance measurements, we should implement all performance affecting features. The most important are packet protection and flow control because they influence performance throughout the lifetime of the QUIC connection. To summarize, the thesis should implement at least the following features:

- Connection lifetime support (establishment, termination)
- Stream multiplexing
- Packet protection
- Loss detection and recovery
- Flow control

On the other hand, we can disregard many QUIC features that react to one-time events or do not otherwise influence the implementation's performance. These features include:

- Connection migration, and therefore multiple Connection IDs support
- Complex (token-based) address validation
- Network path MTU detection
- Version negotiation
- 1-RTT key updates

- Advanced security measures (see Section 21 in transport specification [18])

The rest of the features form a grey area that can be implemented fully, partially, or even not at all if convenient.

3.2 Design Considerations

Before we start the actual analysis, we will briefly outline the design principles used for the actual design of the implementation and their rationale.

3.2.1 Performance

One of the critical factors in the decision between managed .NET implementation of QUIC or using an external library like *MsQuic* is performance. Therefore, the implementation design decisions should focused on greater performance, possibly sacrificing maintainability if the trade-off was justified.

As a general rule, the implementation will:

- *Avoid excessive heap allocations*: Although heap allocation is cheap in .NET, the actual price is paid during garbage collection. The more objects are allocated, the more frequent garbage collections are. Each collection introduces a small stall into the program which could disrupt internal timing of the QUIC implementation. Therefore, heap allocations on hot paths of the code executions should be minimized.

- *Prefer return codes over exceptions*: Throwing an exception is an expensive operation, and their frequent use would have negative impact on the performance.

3.2.2 Testability

The second design aspect we focusd on is the testability of the implementation. Ideally, the design would minimize the need for live debugging of the implementation. This is especially important because stopping the implementation on a breakpoint inevitably disrupts the connection, possibly leading to termination because of idle timeout (see section 2.4.5).

The design intention was to allow writing deterministic automated tests that can inspect the packets sent by the endpoint and verify that they are consistent with the behavior defined by the QUIC specification.

3.3 Target .NET API

The public API, which was designed by the .NET team for use by other developers, uses similar concepts like TCP and UDP .NET networking APIs. There is a `QuicListener` class intended to be used by servers for listening for incoming connections, similarly to using the `TcpListener` class. The actual connection is represented by the `QuicConnection` class, which can be compared to `TcpClient` class. The individual QUIC streams are exposed using the `QuicStream` class, which implements the `Stream` abstraction.

The API is expected to be used asynchronously using the **`async`**/**`await`** model. Most methods return a `ValueTask` which should be **`await`**ed to efficiently wait until the operation completes. The full list of methods with detailed description can be found later in section 5.3.

3.4 High-Level Architecture

The `QuicConnection` implementation will require some sort of background processing thread[1] to send acknowledgments for incoming packets and to resend data after being determined lost due to a timeout. Because the correctness of a multithreaded code is hard to test, our implementation of `QuicConnection` will not interface with the underlying `Socket` instance directly. Instead, the implementation will provide an internal interface for exchanging the datagrams to be sent/received. The actual sending of these datagrams will be handled by a separate class named `QuicSocketContext`. By separating the socket I/O from the connection logic implementation, we can write unit tests that inspect the sent datagrams and assert that their content conforms to the protocol specification.

On the server's side, the `QuicSocketContext` class will also handle any stateless packet processing, such as sending Retry and Version Negotiation packet. It will be also responsible for matching packets to the appropriate `QuicConnection` instances and queueing new connections to `QuicListener` to be read by the application. Figure 3.1 illustrates the relationship between `QuicSocketContext`, `QuicConnection` and `QuicListener` classes.

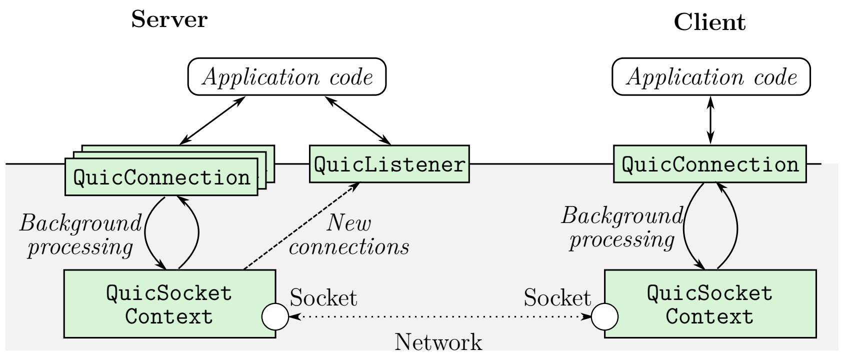


Figure 3.1: High-level background processing architecture.

3.4.1 Servicing the Socket

The `QuicSocketContext` class will implement the necessary background processing management. Its responsibilies are:

- routing incoming datagrams from `Socket` to the proper `QuicConnection` instance;
- calling timeout handlers after a timeout set by the connection expires; and

[1]In order to reduce verbosity, this text will be using the term *thread* to mean both a dedicated `Thread` instance or a `Task` running on a thread-pool thread.

- sending outgoing datagrams provided by the `QuicConnection`.

For performance reasons, it may be better to process timeouts and send the datagrams using one thread and process received datagrams using another thread. However, the cost of synchronization of the shared state may outweigh the performance gain. Our prototype implementation will use a single thread to service both types of events but will allow for the possibility of future experimentation with separate threads for sending and receiving.

Processing Multiple Connections in Parallel

For client connections, the background processing provided by the `QuicSocketContext` class needs to service only a single connection. However, on the server's side, there can be multiple connections receiving datagrams from the same socket and, therefore, being served by a single `QuicSocketContext` instance. Servicing multiple `QuicConnection` instances by only one thread could limit the server's throughput.

In order to allow processing multiple QUIC connections in parallel, our implementation will use a dedicated background `Task` for each `QuicConnection` instance. Our implementation separates the per-connection logic into separate `QuicConnectionContext` class. The `QuicSocketContext` class will be directly responsible only for stateless packet processing like sending a Version Negotiation packet for incoming packets with unsupported versions. Packets belonging to existing connections will be queued for processing by the appropriate `QuicConnectionContext` instance using the `Channel<>` class which provides efficient implemntation of a producer-consumer queue. A possible runtime structure of the `QuicSocketContext` servicing two separate connections is illustrated in Figure 3.2. The two `QuicConnectionContext` instances are serviced by separate dedicated threads. The `QuicSocketContext` itself does not need a dedicated thread. Instead, its logic is performed in whatever thread that completes the pending asynchronous socket receive call.

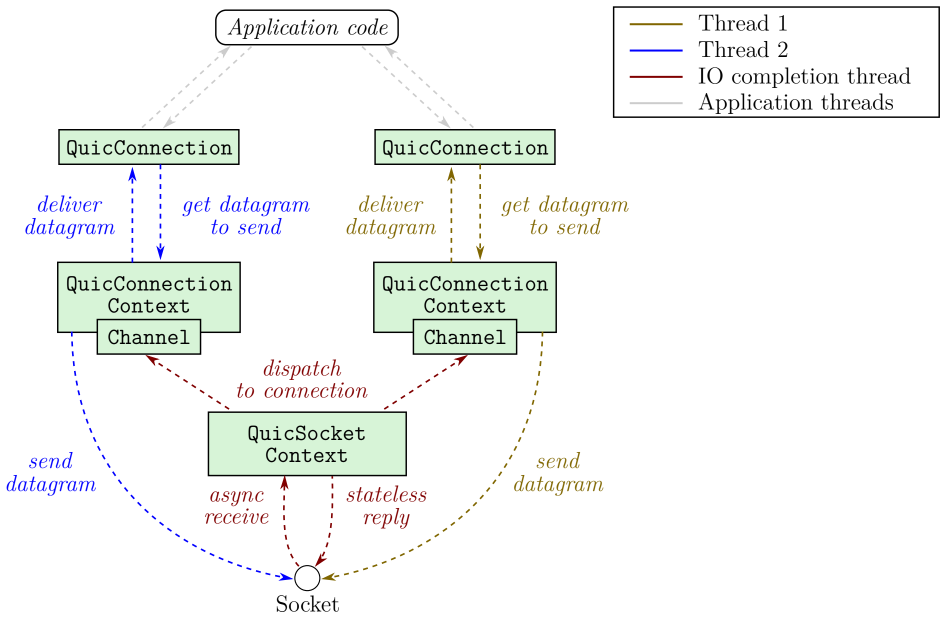


Figure 3.2: Architecture of the server-side background processing.

Future Support for Connection Migration

Lastly, we need to review the architecture for possible support for the connection migration feature. Thanks to the `QuicConnectionContext` not depending directly on a particular `Socket` instance, it is conceivable that a single `QuicConnectionContext` instance could be part of two `QuicSocketContext`s — one for the old endpoint address, and the other for the new one. Use of the `Channel<>` class already ensures that the incoming datagrams could be queued concurrently by both `QuicSocketContext` instances. The only modification needed on the architecture level would be allowing the `QuicConnectionContext` to select the right `Socket` from which the outgoing datagrams should be sent.

3.4.2 Public API Threading model

The target API uses the `ValueTask` type designed for efficient asynchronous methods. The user code will start an asynchronous operation that can be completed by a background thread servicing the connection. This way, the user code cannot block the connection's background thread, which could otherwise cause timeouts to be missed.

The `QuicConnection` can be potentially used in a multithreaded environment, and the implementation should allow concurrent usage of `QuicConnection` and `QuicStream` classes when it makes sense. For example, it makes sense for an application code to process each `QuicStream` in a different thread. However, it does not make sense to concurrently write into one `QuicStream` from two threads without any synchronization. Our implementation will, therefore, provide the following thread-safety guarantees for the API:

- Individual streams can be used concurrently from different threads. However, each direction of the stream (reading and writing) can be accessed only by one thread at a time and must be, therefore, synchronized.
- Accepting/opening new streams on a connection can be done concurrently from multiple threads.
- All other operations on `QuicConnection` must be synchronized, including, e.g., starting and aborting the connection.

3.5 Packet Serialization/Deserialization

Special care needs to be taken when implementing the QUIC packets' serialization to the wire format. Inefficient data representation can negatively impact overall performance because processing individual QUIC packets and frames will likely be a hot path in the implementation. The packet and frame representation should be, therefore, carefully designed to avoid allocations.

The incoming packets can contain arbitrary encoding errors that should be handled efficiently and gracefully, i.e., without throwing exceptions. Possible errors include values being outside of the range of allowed values and incomplete or damaged packets.

3.5.1 QUIC Packet and Frame Representation

Some packets contain many fields. Therefore, passing them around as individual variables would make the implementation harder to maintain. The QUIC frames form coherent messages that should be represented by individual .NET types. This gives us also an opportunity to keep the serialization and deserialization code next to each other, making it easier to ensure that, e.g., the order of the serialized fields matches in both methods.

Representing QUIC frames as individual classes would introduce many heap allocations for every received QUIC packet. This thesis, therefore, will model QUIC frames and the QUIC packet headers as value types. Also, the payload of some frames may consist of large blocks of memory. Examples include `STREAM` and `CRYPTO` frames, which can fill the entire payload of the QUIC packet. Duplicating this block of memory into a separate `byte[]` would be another unnecessary memory allocation. .NET Core 2.1 introduced the `Span<T>` type, which can be used to efficiently reference an arbitrary memory block, including memory allocated on the stack using the `stackalloc` keyword.

The `Span<T>` type is a `ref struct` — a special kind of value type which can be stored only in local variables or inside another `ref struct`s. Limitations on the usage of `ref struct`s also forbid their use as generic type arguments, e.g., for `Func<>` and other generic delegates. This limitation is acceptable for `QuicConnection` implementation because the QUIC frames can be processed one after another right after being deserialized from the QUIC packet.

3.5.2 QuicReader and QuicWriter

Both serialization and deserialization require maintaining the current position in the buffer being read from or written into. In order to simplify the serialization or deserialization code, our implementation introduces `QuicReader` and `QuicWriter` classes as a primary means of reading and writing QUIC primitives to memory. We have chosen `QuicReader` and `QuicWriter` to be reference types allocated on the heap, because we expect to cache and reuse their instances multiple times.

The `QuicReader` and `QuicWriter` classes also take care of converting the endianity of the data between big-endian used by QUIC and the endianity used by the machine running the application. This is done by forwarding the calls to respective methods on the `BinaryPrimitives` class.

3.6 Stream Implementation

QUIC is a transport protocol. Therefore, its entire purpose is transferring streams of data. Since it is likely to be the hot path of the implementation, the internal handling of stream data must be efficient and avoid unnecessary copying of stream data blocks.

QUIC recognizes four types of streams. These streams can have a sending part, receiving part, or both. The fact which endpoint initiated the stream controls only which flow control limits apply to that stream. Otherwise, all streams are handled equally. As mentioned in section 2.5.1, bidirectional streams can be implemented as two unidirectional streams. Therefore, our implementation will divide the QUIC stream logic into `SendStream` and `ReceiveStream` classes, which will handle the sending and receiving behavior of the stream.

3.6.1 Receiving Part of the Stream

The implementation of `ReceiveStream` must buffer the received data before delivering them to the application in case the data are received out-of-order. It also needs to track the amount of buffered memory and how much data was delivered to the application in order to correctly update flow control limits for the peer.

Stream Data Buffering

Ideally, the stream implementation would be structured so that the stream data were copied straight from the decrypted packet to the memory provided by the application. This would be possible if the API used an event-based model with callbacks for incoming data. However, the current API design utilizes a method-based model of the `Stream` abstraction. If the application does not call the `Read` or `ReadAsync` method, there is no application buffer to deliver into.

This implies that in order to avoid intermediate copies of the stream data, the buffer holding the QUIC packet in which the stream data arrived cannot be reused to receive other QUIC packets until the contained stream data are delivered to the application. Tracking which buffers can be reused for receiving the following QUIC packets can be complicated because QUIC packets can carry multiple `STREAM` frames.

We believe that the implementational complexity of avoiding intermediate copies outweighs the possible performance gain. For this reason, our pilot implementation will use a two-copy approach: the first copy from the packet into an intermediate buffer, the second copy from the intermediate buffer into the destination memory provided by the application.

Packet Reordering and Data Deduplication

Unreliability of the UDP protocol can cause QUIC packets to be reordered, lost or received multiple times. Because of that, sections of the stream may be received multiple times, and the contents of `STREAM` frames can arbitrarily overlap.

Even though QUIC Flow control provides an upper limit on the data that needs to be buffered at any given moment, allocating one large buffer may be a waste of memory, as the entire buffer might not be needed at any given point in time. Therefore, our implementation uses a list of smaller buffers and allocates only the necessary number of buffers needed to buffer currently received data. In order to reduce the pressure on the garbage collector (GC), buffers are reused using the `ArrayPool<byte>` class to avoid their frequent allocation.

Reading Data by the Application

Because user code runs on a different thread from the internal `QuicConnection` logic, access to the buffered data must be synchronized. Our implementation uses the Channel<> class, which provides an efficient implementation of the producer-consumer queue with support for asynchronous operations using the ValueTask <> type.

Each time data are received for the stream, the implementation checks if there is a contiguous block of memory that could be delivered to the application. For this reason, the implementation needs to keep track of the parts of the stream which have already been received. If there is a new part of the stream that can be delivered, a view into the relevant region of the buffer is queued into the Channel <> using the Memory<byte> type.

Figure 3.3 illustrates how all parts of the ReceiveStream work together. The data from the incoming STREAM frame are copied to the proper offset in intermediate buffers. These buffers are rented from an ArrayPool<byte> as needed. A Memory<byte> instance providing a view into the newly deliverable parts of the stream is then queued into a Channel<> instance for delivery. The application thread retrieves the memory regions as needed and copies the data into the buffer provided by the application. Once all data from an intermediate buffer are delivered, the intermediate buffer is returned to the ArrayPool<byte> to be reused for buffering future data.

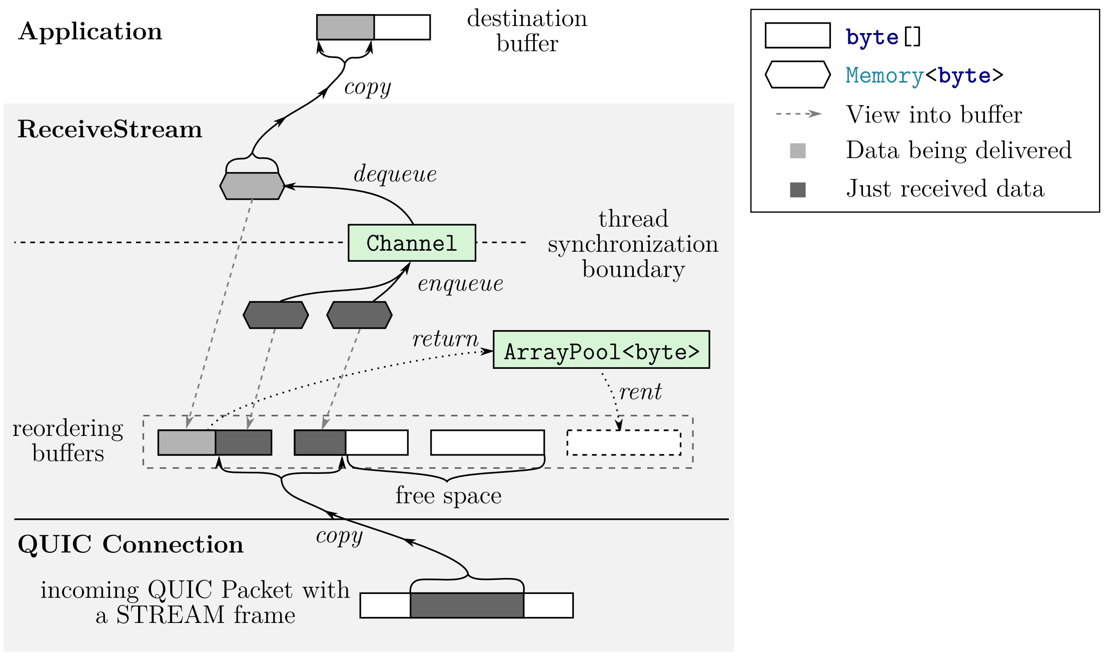


Figure 3.3: Implementation of the receiving part of the stream

Flow Control Considerations

QUIC Flow control limits for streams specify the maximum offset of data that an endpoint can send. An endpoint needs to update these limits by sending a MAX_STREAM_DATA frame after the data are delivered to the application. However, sending updated limits in every packet could waste space that could be used by the application data. On the other hand, updating the limits too late could lead to the other endpoint being blocked and signal the fact by sending a STREAM_-DATA_BLOCKED frame. In such a state, the endpoint cannot send more application data until it receives an update via MAX_STREAM_DATA frame.

Our implementation will send a `MAX_STREAM_DATA` frame after the client uses more than half of the limit provided since the last update. This should prevent updates being too often and at the same time early enough that the sender does not run out of flow control credit.

3.6.2 Sending Part of the Stream

The sending part of a QUIC stream must keep track of the state of all outbound data on the stream. Any part of the stream can be lost during transmission, requiring its retransmission, and any previously sent part of the stream can be acknowledged.

Stream Data Buffering

Similarly to buffering received data, we will analyze how many times the outgoing data need to be copied before they are sent. The semantics of the `Write` and `WriteAsync` methods on the `Stream` class are such that when the method completes, the provided memory can be reused for other purposes.

In this case, a single-copy implementation approach would require that the `Write` and `WriteAsync` methods complete after the peer acknowledges that the data were received. In the best-case scenario where no packet loss occurs, this approach implies a full round trip delay for calling either of the methods. This approach would make the `QuicStream` implementation inconsistent with the other network-enabled `Stream` implementations. Moreover, substantial time would be spent waiting for acknowledgment instead of sending more data. In order to fully saturate the available bandwidth, users of the `QuicStream` would have to either provide very large data buffers or use the `WriteAsync` method and overlap its execution by maintaining multiple outstanding `ValueTask`s.

In order to fit with the other `Stream` implementations, our implementation will create an intermediate copy of the data, and the `Write` and `WriteAsync` will complete once the data is internally buffered. This allows applications to queue enough data to maximally utilize the connection's bandwidth. In order to be memory-efficient, our implementation will, as in `ReceiveStream` implementation, use an `ArrayPool<byte>` to reduce the pressure on the garbage collector.

Acknowledgement, Loss, and Retransmission

Each byte that in the stream can be, conceptually, in three different states:

- *Pending*: The byte needs to be sent to the peer in some future `STREAM` frame.
- *In-flight*: The byte has been sent, but it is uncertain if the containing packet was received.
- *Acknowledged*: The packet containing the byte has been acknowledged by the other endpoint. It is no longer necessary to buffer this byte.

The transitions between these states are straightforward. The data transition from pending to in-flight by sending them in a `STREAM` frame. When the packet is deemed lost by the loss detection algorithm, the data transition back to the pending state, and if the packet has been acknowledged, the data transition to the acknowledged state. The state of the individual parts of the stream can be tracked by maintaining three sets of *ranges* — starts and ends of the blocks of data in the same state.

Writing Data by the Application

Similarly to the receiving part of the stream, our implementation will use `Channel<>` type to provide synchronization with the application thread. However, since `Channel<>` does not allow random access to the provided data, a separate list of buffers is maintained for data which were sent but not yet acknowledged.

Figure 3.4 illustrates the process of writing data into the `SendStream`. The application data are copied to an internal buffer. This buffer comes from a shared `ArrayPool<byte>`. When the buffer is full or the stream is flushed, the buffer is enqueued into the synchronization `Channel<>`. The background thread then dequeues buffers from the `Channel<>` and appends them into a list of retransmission buffers. Each time a new `STREAM` frame for this stream is written into a QUIC packet, the earliest not-yet-sent data are selected and copied into the outgoing QUIC packet buffer. Once all data in a particular retransmission buffer are acknowledged, the buffer can be removed and returned to the `ArrayPool<byte>`.

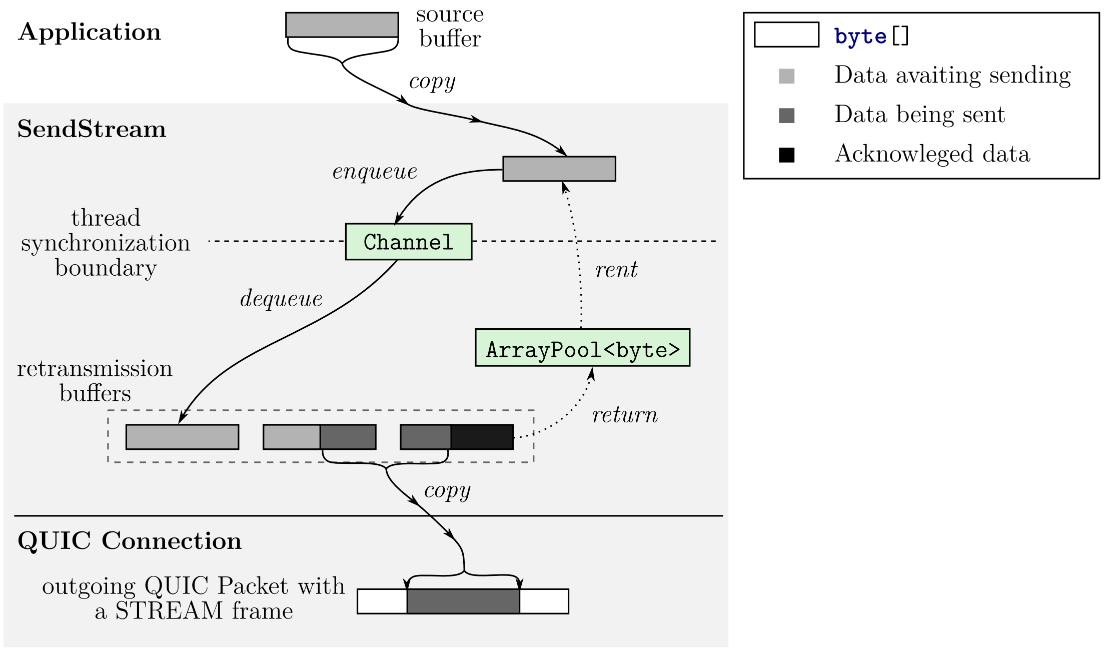


Figure 3.4: Implementation of the sending part of the stream

3.6.3 Abort/Dispose Model for Streams

The `QuicStream` class implements the `IDisposable` interface which should provide automatic closing of the stream when `QuicStream` is used in a **`using`** statement or **`using`** block. This implies resetting the writable part of the stream using `RESET_STREAM` frame and requesting reset of the readable part of the stream using `STOP_SENDING` frame.

However, both these frames require an application-level error code. QUIC specification does not define any transport-level error codes for these stream operations and there is no way for the application to specify which error code should be used. This has been established as a flaw of the current API and is expected to change in future iterations design iterations [26].

Leaving `IDisposable` unimplemented would be misleading to users in the prototype implementation. Therefore, our implementation will deviate from the specification by using error code 0 regardless of its semantics in the application protocol, unless the stream has been explicitly closed by the application using the `AbortRead` or `AbortWrite` methods.

Another area where the QUIC API is not well defined is the behavior of pending `AcceptStream` calls on `QuicConnection` when the connection is closed. In case the connection has been closed gracefully from the application protocol's perspective, it would be better if this method did not throw an exception but returned **`null`** reference. However, the semantics of the error codes are defined by the application protocol and, therefore, the QUIC implementation cannot distinguish between graceful or abortive connection close. Until the behavior of these methods in such scenarios is better defined, our implementation will throw `QuicConnectionAbortedException` for all pending async calls.

3.7 TLS Implementation

TLS handshake forms an integral part of the QUIC connection establishment. Because correct TLS implementation is crucial for ensuring the security of the resulting implementation, this thesis should avoid implementing TLS algorithms by itself. Instead, it should reuse some existing and well-tested implementation.

The novel way QUIC integrates with TLS requires a specific API from the TLS implementation. Below is a non-exhaustive list of operations the TLS library's API must provide:

- Querying the current state of the handshake
- Retrieving both the application-level secrets and secrets used to protect the handshake process
- Retrieving raw, unencrypted TLS messages to be sent to the other endpoint
- Obtaining the negotiated cipher
- Specifying protocols used for ALPN
- Specifying a custom TLS extension in order to exchange QUIC transport parameters

The .NET runtime libraries use different native libraries to provide TLS functionality on different operating systems. On Windows, .NET uses ***Secure Channel*** [27] (***Schannel*** for short) which is part of the Windows operating system. On Linux and macOS systems, the ***OpenSSL*** library [28] is used.

Secure Channel

The *Schannel* versions present in the latest Windows 10 builds support only TLS 1.2. However, future updates will also implement TLS 1.3. A preview of the *Schannel* with TLS 1.3 support can be obtained by installing an Insider build of Windows 10. Because *MsQuic* uses the *Schannel* library when compiled for Windows, we assume that *Schannel* exposes the API necessary for a QUIC implementation.

OpenSSL

None of the mainstream versions of *OpenSSL* library expose necessary API for integration into QUIC, and there are no plans to include such API in the next *OpenSSL* 3.0.0 release [29]. However, developers at Akamai maintain a fork of *OpenSSL* which adds the QUIC-enabling API [30]. This modified version of *OpenSSL* is used by *MsQuic* (in Linux builds) and some other QUIC libraries like *quiche* from Cloudflare [31]. Akamai's changes may be merged into *OpenSSL* for the following 3.1.0 or later releases.

In conclusion, the APIs required for our QUIC implementation are currently only accessible in only the preview versions of Windows 10. Relying solely on *Schannel* for TLS 1.3 support would severely impact cross-platform availability of our prototype implementation. We have, therefore, decided to integrate with the modified *OpenSSL* library which supports all platforms supported by .NET. This solution, however, has some drawbacks:

- The modified *OpenSSL* binary must be present on the machine running our QUIC implementation. This implies that the library must be compiled from source beforehand for the target machine because pre-built binaries for the modified library are not available.
- Only a limited integration with X.509 certificates is possible because the `X509Certificate` class implementation will be using a different binary — *CryptoAPI* on Windows, unmodified *OpenSSL* on Linux and macOS.

These drawbacks are acceptable for the prototype implementation and will be eliminated once the support for QUIC is released in mainstream versions of the *OpenSSL* and *Schannel* libraries.

3.8 Packet Protection

As described in section 2.8.2, the packet encryption process consists of two phases — payload protection and header protection. The combined process requires the following inputs:

- Protection keys derived using the process explained in section 2.8.2
- Negotiated cipher
- Packet number (for encryption), or expected packet number (for decryption)
- The QUIC packet to encrypt or decrypt

The cipher is negotiated once and cannot be changed during the lifetime of the connection. Protection keys can be changed only for the 1-RTT packets using the process of *key update* (see section 2.8.2), which can be expected to be relatively infrequent. The rest of the inputs change with every packet. Therefore, our implementation encapsulates the packet protection implementation in a `CryptoSeal` class, which does not depend on the rest of the QUIC implementation.

The process of receiving packets requires intermediate validation of the header fields. The individual steps — header protection and payload protection — must be, therefore, exposed separately. Also, the encryption and decryption should happen in-place to avoid unnecessary allocations and copying of the packets.

Important consideration needs to be made for performing the actual encryption/decryption once all inputs to AEAD have been gathered. .NET does not contain an implementation of the CHACHA family of ciphers. Fortunately, the *OpenSSL* library can be configured to not allow this cipher to be negotiated and, therefore, our prototype can work without the CHACHA cipher support.

The other ciphers are based on the AES family of ciphers, which are supported by .NET. However, individual classes implementing these ciphers do not share a common base class or interface. Therefore, our implementation wraps the concrete AES implementations in classes derived from an abstract `CryptoSealAlgorithm` class which defines a common interface required by the `CryptoSeal` class. Figure 3.5 illustrates how the `CryptoSeal` and `CryptoSeal Algorithm` classes are connected.

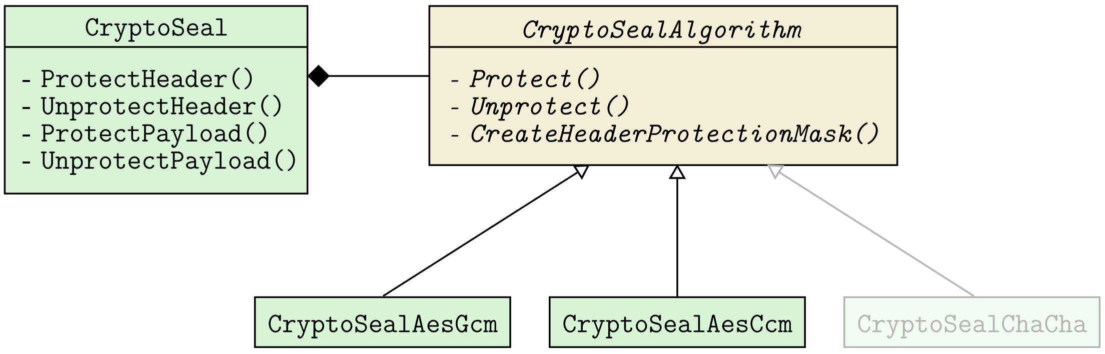


Figure 3.5: Relationship between CryptoSeal and CryptoSealAlgorithm classes

The key update operation can be implemented by replacing the existing instance of the `CryptoSeal` class with a new one with the updated protection keys.

3.9 Loss Detection and Loss Recovery

In order to detect lost packets and retransmit any lost data, the `QuicConnection` implementation must keep track of which data have been sent in which packets and the timestamp when the packet was sent for timeout detection. Our implementation will encapsulate this information and the loss detection algorithm in a dedicated `RecoveryController` class, which does not depend on the `Quic Connection` class. This way, its implementation can be unit tested separately from the rest of the connection logic.

Another responsibility of the RecoveryController class is maintaining the congestion window. Although the specification defines only a single algorithm for congestion control based on TCP NewReno [20, Section 7], there are already experiments with other algorithms like HyStart++ and CUBIC [32]. The ability to support multiple such algorithms could provide an opportunity for future experimentations. For that reason, the RecoveryController will use the strategy pattern [33] to allow choosing the congestion control algorithm at runtime.

3.10 Automated Testing

It is necessary to implement automated tests that assert that the implementation conforms to the QUIC protocol specification. Additionally, the public API makes use of existing concepts, namely the Stream abstraction for QUIC streams, and thus it is necessary to ensure that QuicStream conforms to the expected Stream behavior.

The .NET runtime repository into which we wish to integrate our QUIC implementation uses the ***xUnit*** testing framework [34]. The ***xUnit*** framework is one of the most mature testing frameworks for .NET. Therefore, we will use it as well for writing tests for our implementation.

3.10.1 Testing the QUIC Protocol

The QUIC protocol specification [18] mostly specifies the protocol behaviors in terms of what endpoint may or may not send in specific scenarios. This implies that the correctness of the implementation as a whole can be tested by inspecting the contents of the generated UDP datagrams.

Because our implementation separates QUIC connection logic from the socket IO, the unit tests can be written against the internal QuicConnection API. Exchange of QUIC packets between two QuicConnection instances can be simulated by the unit testing code. An advantage of using this internal API is that the unit tests can be written as single-threaded — without any background processing thread which could introduce nondeterminism to the tests. However, in order to make the tests completely deterministic, the testing code also must be able to control the exact timing of events. In order to achieve that, our QuicConnection implementation will not obtain the current timestamp directly, e.g., by calling the Stopwatch.GetTimestamp() method. Instead, the current timestamp will be provided as an argument to the internal API, either by the QuicConnection Context class or the unit testing code.

Testing Harness

A large number of the unit tests will consist of inspecting contents of the QUIC packets exchanged between a pair of QuicConnection instances or checking the internal state of a connection after a particular packet exchange. However, correctly implemented QuicConnection will never violate the protocol and, therefore, additional logic is required to test error conditions.

In order to test the behavior in erroneous conditions, the testing code needs to be able to either compose an invalid packet to be sent or intercept a valid packet and modify it to elicit an error response. However, doing this is difficult for two reasons:

- All packets are encrypted and protected against modification. The testing code must first retrieve the correct `CryptoSeal` from the sender `QuicConnection` instance and unprotect the packet. After any modification, the encryption must be reapplied.
- Modifying a particular QUIC frame may change the encoded frame's size because of the variable-length encoding used (see section 2.1.3). If the size of the QUIC frame changes, then all following frames must be shifted, and the Length field in the packet header must be adjusted.

Doing this manually would make unit testing function very verbose and make the code hard to understand. To keep the tests concise and easy to understand, we will implement a *testing harness* which will provide the functionality mentioned above via a set of helper methods. These methods will provide a declarative way to specify what the QUIC packet or datagram must contain and register callbacks for packet modification to elicit error responses.

Listing 3.1 shows an example how the desired testing harness would be used in conjunction with ***xUnit*** testing framework to test if the first UDP datagram sent by the client has the correct size[2]. The `GetDatagramToSend` method will get the next UDP datagram to be sent and present it in a structured manner. Later, the `ShouldHaveFrame<TFrame>` method will internally check if a frame represented by `TFrame` type is present in the packet and will invoke the provided callback for further assertions.

Listing 3.1: Example unit test inspecting QUIC packets contents.

```
[Fact]
public void ClientInitialDatagramHasInitialPacketWithCryptoFrame()
{
    var datagram = GetDatagramToSend(Client);
    Assert.Equal(
        QuicConstants.MinimumClientInitialDatagramSize,
        datagram.Size);

    var initial = Assert.IsType<InitialPacket>(
        Assert.Single(datagram.Packets));
    Assert.Equal(0, initial.PacketNumber);

    initial.ShouldHaveFrame<CryptoFrame>(crypto =>
    {
        Assert.Equal(0, crypto.Offset);
        Assert.NotEmpty(crypto.CryptoData);
    });
}
```

However, in section 3.5.1, we mentioned that the types representing QUIC frames must be **`ref structs`** in order to contain `Span<T>` instances, and that

[2]As part of the prevention against traffic amplification attacks, all UDP datagrams containing an Initial packet sent by client must be larger than 1200 bytes [18, Section 8.1].

`ref struct`s cannot be used as type arguments for generic classes or methods. This could be overcome by using `Memory<T>` instead of `Span<T>`, but the use of structs for frame types still poses a problem for modification of the QUIC frames by a callback.

Structs are normally passed by value; therefore, the callback would either have to accept the frame by reference using the `ref` keyword or return the modified frame as the return value. Neither of these solutions is perfect. Parameters types with `ref` modifiers cannot be inferred for lambdas and need to be stated explicitly, and having to return the modified frame is unintuitive for non-modifying callbacks. Furthermore, `struct`s representing QUIC frames should be preferably made `readonly` to allow more compiler optimizations, so modifying the frame would require creating a new instance of the frame, overwriting the old value.

For the above reasons, we decided to duplicate the types for frame representation using mutable `class`es. This duplication will allow expressing the unit tests in a succinct, natural declarative manner. This is illustrated in the test in Listing 3.2 which uses an `Intercept1RttFrame<TFrame>` method to intercept a frame of type `TFrame` and modify it. In this case, we shift the range of acknowledged packets by one and, by doing so, simulate acknowledging a packet that the server has not sent yet, which is a violation of the protocol which should result in connection termination via a CONNECTION_CLOSE frame.

Listing 3.2: Example unit test simulating error conditions.

```
[Fact]
public void ConnectionCloseWhenAckingFuturePacket()
{
    // ... setup ommited for brevity

    Intercept1RttFrame<AckFrame>(Client, Server, ack =>
    {
        // ack one packet more than originally intended
        ack.LargestAcknowledged++;
    });

    Get1RttToSend(Server).ShouldHaveFrame<ConnectionCloseFrame>(f =>
    {
        Assert.Equal(TransportErrorCode.ProtocolViolation, f.ErrorCode);
        Assert.Equal(FrameType.Ack, f.ErrorFrameType);
        Assert.Equal(QuicError.InvalidAckRange, f.ReasonPhrase);
    });
}
```

3.10.2 Testing the Public API

The .NET runtime repository already contains a suite of tests used to test the *MsQuic*-based QUIC implementation. Another large separate suite of tests exists for ensuring that all `Stream` implementation behave consistently. All these tests can be used to ensure that our implementation of `QuicListener`, `QuicConnection`, and `QuicStream` classes conforms to the public API specification.

3.11 Diagnostics

QUIC is a very complex protocol, and bug investigation can be complicated. By stopping the application on a breakpoint, the developer inadvertently changes the application's behavior. Extended pauses for inspecting internal connection state make the implementation miss important timeouts, possibly leading to connection being closed due to the idle timeout. This makes interactive debugging of issues which span multiple roundtrips almost impossible.

There are two possible ways of gaining better insight into the connection's behavior — externally inspecting the packets sent over the network and producing verbose logs (called traces) by the implementation.

3.11.1 Inspecting the Sent Packets

Inspecting the connection behavior by observing the packets sent over the network is a non-invasive way of diagnosing issues in any networking protocol. An example of a tool that can be used for this task is Wireshark [35], which also supports QUIC.

However, since QUIC is always encrypted, inspecting QUIC packets via Wireshark is not as straightforward as for other network protocols like plain TCP. Implementations must leak the encryption secrets used in the connection, e.g., into a log file, and these keys must be provided to Wireshark to enable decrypting the packets. While this approach is relatively simple to implement, it provides little concrete information about the internal state of the connection. The developer must infer the internal connection state from his knowledge about the implementation and the contents of the captured QUIC packets.

3.11.2 Producing Traces from the Connection

A better insight into the connection's behavior can be gained by emitting verbose logs that can be later analyzed. However, such traces can consist of thousand lines of output which may be difficult to read and reason about. Fortunately, there are tools which can visualize individual events in the connection in a graphic format that is easier to understand. One of these tool is *qvis* [36] which is part of the *quiclog* suite [37]. The *qvis* tool consumes logs in a JSON format, which can be produced directly or generated from implementation-specific log format or even from generated from network traffic captured using Wireshark, provided that the encryption secrets used in the connection are known.

For our implementation, we have decided to keep the tracing implementation simple by directly emitting traces in the JSON format which can be directly consumed by the *qvis* tool without any further conversions. However, serializing packet information into JSON format is likely to incur noticeable overhead and, therfore, a better logging format should be considered for future development.

3.12 Integration into .NET Runtime Codebase

The work of this thesis aims to be eventually mergeable into the .NET runtime codebase. As of writing this thesis, there already exists a project for *Sys-*

tem.Net.Quic.dll. Therefore, the source code may be placed there directly without additional changes.

There is, however, the issue with the ***OpenSSL*** dependency. The ***OpenSSL*** library is written in C and must be compiled for the target machine. Integrating the ***OpenSSL*** compilation into the .NET runtime build process would impose additional requirements for the .NET runtime compilation. The additional dependencies would complicate building the .NET runtime as part of the continuous integration process. For this reason, it is necessary to provide also a mock TLS implementation that would be used in automated tests as part of continuous integration.

The mock TLS implementation can also be used locally to avoid building the ***OpenSSL*** dependency. The only limitation of the mock TLS implementation is that it does not allow interop with other QUIC implementations.

4. Developer Documentation

The Managed QUIC implementation developed in this thesis is contained in a fork of the official .NET runtime repository. The source code of this fork is attached in `src/dotnet-runtime/` directory in the thesis attachments.

The documentation inside the .NET runtime repository contains detailed workflow instructions necessary for the development of the .NET runtime. These instructions list the necessary prerequisites and explain how to build the product and run unit tests. The workflow instructions' top-level file is located at `docs/workflow/README.md`.

This chapter will focus on the `System.Net.Quic` library, which contains the QUIC protocol implementation. The source code for this library is located inside the `src/libraries/System.Net.Quic/` directory inside the .NET runtime codebase. The directory listing in Listing 4.1 shows the structure of the `System.Net.Quic` project directory. The listing also emphasizes the `Managed` and `UnitTests` directories, which contain the code developed as part of this thesis. These directories are the main focus of this chapter.

Listing 4.1: Directory structure of the System.Net.Quic project. Emphasised items contain the implementation developed in this thesis.

```
src/dotnet-runtime/src/libraries/System.Net.Quic/
├─ ref ..................................................... Refererence assembly code
├─ src ..................................................... Main library source code
│  ├─ Resources
│  │  └─ Strings.resx ........... Definition of localizable strings like exception messages.
│  ├─ System
│  │  └─ Net
│  │     └─ Quic
│  │        ├─ Implementations ............. Root directory for all QUIC implementations
│  │        │  ├─ Managed ........................... This thesis' implementation sources
│  │        │  ├─ MsQuic ........................... MsQuic-based implementation sources
│  │        │  └─ Mock ........................... Mock implementation used only in tests
│  │        └─ Interop ..................................... Imports from native libraries
│  └─ System.Net.Quic.csproj
├─ tests ................................................... Library tests source code
│  ├─ certs ................................................ X.509 certificates used in tests
│  │  ├─ cert.crt .......................................... Public certificate file
│  │  └─ cert.key .......................................... Private key file
│  ├─ FunctionalTests ................................... Tests against the public API
│  │  └─ System.Net.Quic.Functional.Tests.csproj
│  └─ UnitTests .................................... Managed implementation unit tests
│     └─ System.Net.Quic.Unit.Tests.csproj
├─ Directory.Build.props
└─ System.Net.Quic.sln
```

4.1 QUIC Implementation Providers

The `System.Net.Quic` project internally contains an infrastructure for switching between multiple implementations of the QUIC protocol. The API classes themselves are **`sealed`** but they delegate all methods to polymorphic *implementation providers*. Each `QuicListener`, `QuicConnection`, and `QuicStream` instance contains a reference to a `QuicListenerProvider`, `QuicConnectionProvider`, or `QuicStreamProvider` instance, respectively. Implementations of each provider are provided by each QUIC implementation in the `System.Net.Quic` library. Figure 4.1 illustrates this indirection layer using a class diagram.

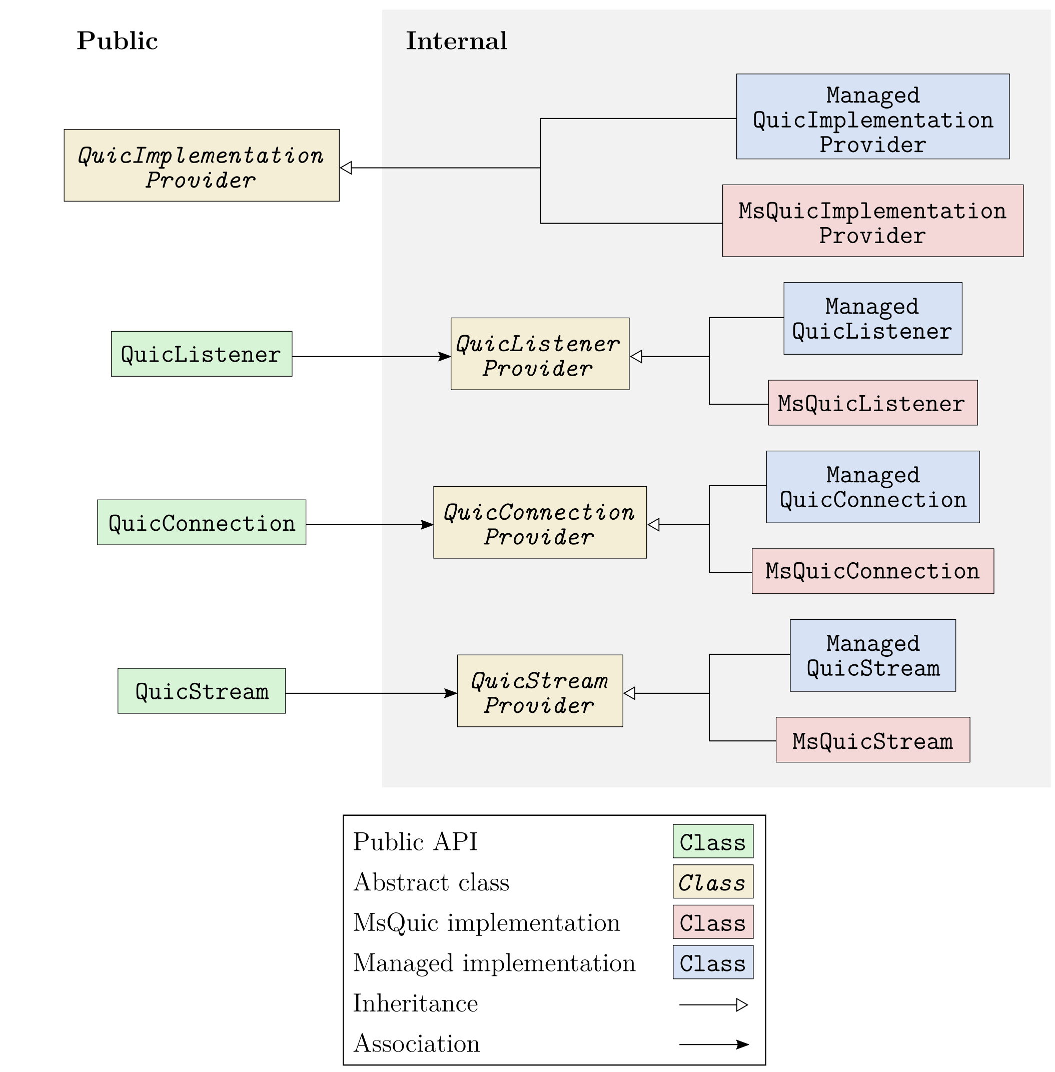


Figure 4.1: Implementation providers for the QUIC API classes

The infrastructure implements the abstract factory pattern [38]. New instances of `QuicListenerProvider` and `QuicConnectionProvider` are created by concrete implementations of the `QuicImplementationProvider` class. The singleton instances of `QuicImplementationProvider` class implementations are exposed as static properties on the `QuicImplementationProviders` static class.

There were two pre-existing `QuicImplementationProviders` in the `System.Net.Quic` project. Implementation of these two providers is not the primary focus of this text and, therefore, this text will not provide further details on these providers:

- `MsQuic`: QUIC implementation backed by ***MsQuic*** native library
- `Mock`: A mock QUIC implementation for use in tests

As part of this thesis, we implemented the following two new providers:

- `Managed`: Managed implementation with TLS backed by ***OpenSSL*** fork with QUIC enabling API.
- `ManagedMockTls`: Managed implementation with mock TLS implementation, which does not depend on external libraries, but cannot interoperate with other QUIC implementations.

Additionally, there is the `Default` provider. This provider can be influenced by setting the `DOTNETQUIC_PROVIDER` environment variable to the desired provider name. If the environment variable is not set, then the `Managed` provider is used.

The `QuicListener` and `QuicConnection` classes have a constructor overload which accepts an instance of the `QuicImplementationProvider` to be used. This way, the QUIC implementation can be selected during runtime. This also allows reusing a suite of functional tests for all implementations by simply changing the implementation provider.

4.2 Managed QUIC Implementation Overview

The source code for the managed implementation developed in this thesis is located under the `System.Net.Quic/src/System/Net/Quic/Implementations/Managed/` subdirectory. Listing 4.2 outlines the directory structure of the implementation.

Listing 4.2: Directory structure of the managed QUIC implementation

```
System.Net.Quic/src/System/Net/Quic/Implementations/Managed
 ├─ Internal ......................... Internal code of the implementation
 │   ├─ Crypto ............................... Cryptographic facilities
 │   ├─ Frames ............................ Definition of QUIC frames
 │   ├─ Headers ................... Definition of QUIC packet headers
 │   ├─ Packets ................. QUIC packet number spaces handling
 │   ├─ Parsing ......................... Parsing of QUIC primitives
 │   ├─ Recovery ....................... Loss detection and recovery
 │   ├─ Sockets .............................. Servicing socket IO
 │   ├─ Streams ................................ Stream buffering
 │   ├─ Tls ...................................... TLS integration
 │   │   ├─ Mock ............................ Mock TLS implementation
 │   │   └─ OpenSsl ......................... OpenSSL TLS integration
 │   └─ Tracing ....................... Tracing and logging facilities
 └─ ManagedQuicConnection.cs ............ Implementation of public API
```

```
│
├─ ManagedQuicConnection.Frames.cs ...................... Processing of QUIC frames
├─ ManagedQuicConnection.Packets.cs .................... Processing of QUIC packets
├─ ManagedQuicConnection.Recovery.cs ......................... Handling packet loss
├─ ManagedQuicConnection.Stream.cs ........................... Stream management
├─ ManagedQuicImplementationProvider.cs ........... Abstract factory implementation
├─ ManagedQuicListener.cs ............................. Implementation of public API
└─ ManagedQuicStream.cs ............................... Implementation of public API
```

The implementation is exposed using the `ManagedQuicListener`, `ManagedQuicConnection`, `ManagedQuicStream` implementation provider classes and the `ManagedQuicImplementationProvider` factory. The source code for these classes can be found in the root directory of the implementation.

The high-level architecture has been described in section 3.4. This and following sections will provide further implementation details. The class diagram in Figure 4.2 shows the releationship between the key architecture classes which were also mentioned in Figure 3.1 and Figure 3.2. These classes are:

- `ManagedQuicConnection`: Implementation provider for `QuicConnection`. Implements stateful connection logic.
- `ManagedQuicListener`: Implementation provider for `QuicListener`. Maintains a queue of incoming connections to be accepted by the application.
- `QuicConnectionContext`: Class hosting the background thread for servicing a single `ManagedQuicConnection`, including timeout expiration and sending or receiving UDP datagrams with QUIC packets.
- `QuicSocketContext`: Abstract class handling basic sending and receiving of UDP datagrams, base class for `QuicClientSocketContext` and `QuicServerSocketContext`.
- `QuicServerSocketContext`: Implements server-side stateless UDP datagram processing and dispatch of incoming UDP datagrams to appropriate `QuicConnectionContext` instance.
- `QuicClientSocketContext`: Implements client-side `QuicSocketContext` behavior, passing all packets to a single single `QuicConnectionContext`.

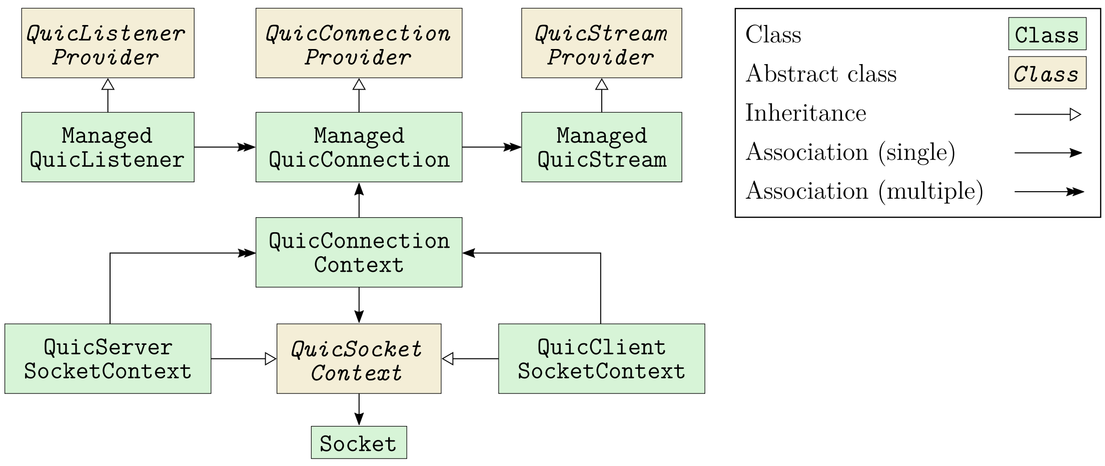


Figure 4.2: Relationship between key classes of managed QUIC implementation

4.3 Supporting Data Structures

The QUIC implementation requires a few specialized data structures for internal implementation.

4.3.1 RangeSet

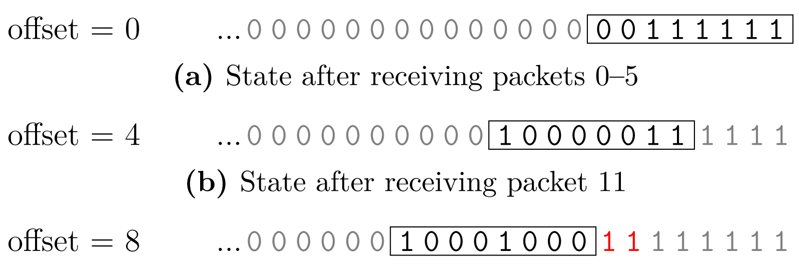
Efficient representation of ranges of acknowledged packet numbers and maintaining information about which parts of the QUIC stream have been sent or acknowledged require a data structure capable of efficiently performing set operations on ranges of integers. The `RangeSet` class represents such a set.

It is expected that the number of ranges in one `RangeSet` instance will be relatively small. Therefore, our implementation uses a `List<T>` instance to store individual ranges which are sorted in ascending order. The ranges themselves are represented by `RangeSet.Range` struct which contains the first and last element represented by the range.

4.3.2 PacketNumberWindow

The `PacketNumberWindow` class is used to test if QUIC packet with the given packet number has already been received. Internally, it contains two 64-bit **`ulong`** fields: a bitmask marking the received packet numbers and the offset of the window. All packet numbers above the window are considered not received, and all packet numbers below the window are considered received. Packet numbers inside the window are considered received if the corresponding bit is set to 1. When a new packet number is received, the window is shifted, if necessary, and the corresponding bit is set. This allows tracking 64 consecutive packet numbers at any given moment with very low overhead.

Figure 4.3 illustrates the concept with an smaller 8-bit window. Figure 4.3a shows the state of the `PacketNumberWindow` after receiving packets 0–5. When packet 11 is received, the window is shifted by 4 and the highest bit — which now corresponds to the packet number 11 — is set to 1, as shown in Figure 4.3b. Figure 4.3c shows the state of the window when packet 15 is received next. The window needs to be shifted by another 4 bits, moving unreceived packet numbers 6 and 7 outside the window (marked by red color in the figure).



(a) State after receiving packets 0–5

(b) State after receiving packet 11

(c) State after receiving packet 15

Figure 4.3: Maintaining window of received packet numbers

The packet numbers marked in red in Figure 4.3c demonstrate that the data structure may falsely label some old packet numbers as already received. In such a case, the packets are simply discarded by the `ManagedQuicConnection` implementation. This may be acceptable for the following reasons:

- The discarded packet will not be acknowledged, and the data will, therefore, be eventually retransmitted in some other packet. The only possible harm to the connection will come in the form of degraded performance because the other endpoint will interpret it like a packet loss and will reduce his congestion window.
- For a packet N to be falsely discarded in this way with a 64-bit packet window, it must have been delayed long enough for packet $N + 64$ or newer to be received first.
- Even if we used an exact method to track received packet numbers and correctly received packet N, it is almost sure that it already was or will be marked as lost on the sender's side. This is because, by that point, an acknowledgment had already been sent for packet $N + 3$ or higher, which will cause the packet to be considered lost due to the packet reordering threshold[1].

Returning to the situation illustrated in Figure 4.3. When packet 11 is received (Figure 4.3b), an acknowledgment is immediately sent. Once the sender receives the acknowledgement — which may be before or after the situation from Figure 4.3c — it will mark packets 6 and 7 as lost and resend the data in some future packets.

4.4 ManagedQuicConnection Implementation

The `ManagedQuicConnection` class implements the stateful QUIC connection logic, which makes up most of the managed QUIC implementation. The source code of the `ManagedQuicConnection` class is separated by area into multiple files, as outlined previously in Listing 4.2:

- *Public API*: Implementation of the public API methods inherited from the `QuicConnectionProvider` class.
- *Packets*: Processing individual QUIC packets, applying and removing the packet protection, generating packets to be sent.
- *Frames*: Processing individual QUIC frames and generating frames for outgoing packets.
- *Stream*: Management of created QUIC streams and flow control limits.
- *Recovery*: Tracking of sent QUIC packets, handling of acknowledgments and packet loss. Congestion window management.

The following subsections describe the major parts of the implementation.

[1]Receiving an acknowledgment for packet N will mark all packets $N - \mathit{kReorderingThreshold}$ or lower as lost. The QUIC specification recommends value 3 for the *kReorderingThreshold* constant. The QUIC loss detection algorithm has been described in greater detail in section 2.7

4.4.1 Integration with Socket Management

The `ManagedQuicConnection` implementation is separated from socket IO management. Also, to allow for deterministic unit testing, the implementation does not maintain an internal timer that would automatically invoke some logic on expiration. Instead, all interactions with a `Socket` are driven externally by a `QuicConnectionContext` class that maintains a background processing thread for handling timeouts and sending and receiving QUIC packets. The interface used by `QuicConnectionContext` consists of following members on `ManagedQuicConnection`:

`QuicConnectionState ConnectionState { get; }`
: The current state of the connection. Used to detect transitions to, e.g., connected state or closed state.

`void SendData(QuicWriter, out EndPoint, QuicSocketContext.SendContext)`
: Allows the connection to write a UDP datagram into the `QuicWriter` instance and specify the `EndPoint` to which the UDP datagram should be sent. The `SendContext` instance contains additional data like the current timestamp.

`void ReceiveData(QuicReader, EndPoint, QuicSocketContext.RecvContext)`
: Processes a datagram from the provided `QuicReader` instance. The `RecvContext` instance contains additional data like current timestamp.

`long GetNextTimerTimestamp()`
: Retrieves the timestamp when the next internal timer of expires. Examples of such timers are loss detection timer, draining timer before closing the connection, or pacing timer, which evens out the outbound packet flow. When the timer expires, the `ManagedQuicConnection` instance may have more data to send.

The value of `GetNextTimerTimestamp` is then used to suspend the background processing thread in order not to consume CPU resources. The background processing thread waits until either the timer expires or until a new QUIC packet arrives. However, some application code actions like writing data to stream require interrupting the wait. This is achieved by calling the `QuicConnectionContext.WakeUp()` method from the `ManagedQuicConnection`.

4.4.2 Managing Packet Number Spaces

The `ManagedQuicConnection` class maintains an internal array of three `PacketNumberSpace` instances which encapsulate all state relevant for individual packet number spaces. The data maintained in the `PacketNumberSpace` include:

- next packet number to be sent,
- `PacketNumberWindow` of received packet numbers,
- largest received packet number and timestamp when the packet was received,

- RangeSet of received packet numbers that are not yet acknowledged,
- whether an ack-eliciting packet was received and an ACK needs to be sent,
- CryptoSeal instances for protecting and unprotecting QUIC packets, and
- SendStream and ReceiveStream for buffering cryptographic data from TLS to be sent in CRYPTO frames.

The data in PacketNumberSpaces is updated by the ManagedQuicConnection each time a QUIC packet is sent or received.

4.4.3 Packet Loss Detection, Recovery and Congestion Control

The implementation of loss detection and recovery are delegated to Recovery Controller class to allow easier testing. Each sent packet is represented by an instance of SentPacket which contains information about the packet which is relevant to the loss detection and recovery algorithms, such as packet number, the timestamp when the packet was sent, packet number ranges acknowledged by the packet, size of the packet, and list of data ranges sent in STREAM frames.

Similarly to PacketNumberSpace class used to maintain connection-wide state for each packet number space, the RecoveryController maintains an array of RecoveryController.PacketNumberSpace instances which contain about each packet number space that are relevant only for recovery purposes. These include:

- largest packet number acknowledged by the peer,
- timestap of last ack-eliciting packet sent,
- SentPackets awaiting acknowlegement,
- SentPackets that are newly acknowledged, and
- SentPackets that are newly considered lost,

Additionally, the RecoveryController maintains some data that is shared across all packet number spaces. These are mostly data relevant for congestion control algorithm and *packet pacing*. Packet-pacing is a mechanism that evens out outgoing packets to prevent *micro-bursting* — a phenomenon in which packets arrive in short rapid bursts which may overflow the receiver and cause packet loss. The data managed by RecoveryController itself include:

- estimates of the current round trip time,
- timestamp of the next packet loss event,
- timestamp when the last UDP datagram was sent,
- size of the last sent UDP datagram,
- number of bytes currently in-flight,

- current size of the congestion window, and
- `ICongestionController` instance implementing the selected algorithm for congestion control.

The main interface methods exposed to the `ManagedQuicConnection` implementation are:

`long LossRecoveryTimer { get; }`
: Timestamp when the next packet loss will occur unless an `ACK` from peer is received.

`void OnLossDetectionTimeout()`
: Performs loss detection and populates collections of `SentPacket` instances on appropriate `PacketNumberSpace` instance with packets which are now considered lost.

`void OnPacketSent(PacketSpace space, SentPacket packet)`
: Registers the `SentPacket` instance as sent and tracks it in loss detection algorithm.

`void OnAckReceived(PacketSpace space, RangeSet acknowledged)`
: Acknowledges packet numbers from provided `RangeSet` and moves appropriate `SentPacket` instances the collection of newly acknowledged packets.

`int GetSendingAllowance(long timestamp)`
: Gets the maximum size of a UDP datagram that the pacer will allow to be sent at the given timestamp.

`long GetPacingTimerForNextFullPacket()`
: Gets timestamp when the pacer will allow sending next QUIC packet of maximum size[2].

4.4.4 QUIC Stream Management

The management of QUIC streams is delegated to `StreamCollection` class. The `StreamCollection` tracks already created streams by their type and checks that neither endpoint exceeds the maximum number of created streams. It also implements efficient lookup of `ManagedQuicStream` instances by their Stream IDs and tracks queues of streams with QUIC frames to be sent in the following QUIC packets.

The `StreamCollection` class maintains two independent queues of `ManagedQuicStream` instances:

- *Flushable*: Streams that have data to send within the flow control limits on that stream.

[2]The maximum size of an outgoing QUIC packet depends on multiple factors. Most significantly, it must be small enough to avoid fragmentation of the UDP datagram by lower network layers. The other endpoint can also set an upper limit on the UDP datagram size it is willing to receive. This limit is provided using the `max_udp_payload_size` transport parameter during the connection handshake.

- *Updateable*: Streams for which some other than the STREAM frame needs to be sent. This includes updating flow control limits or aborting the stream.

The reasoning for a separate queue for flushable streams is that when composing a packet, STREAM frames should be the last frames written into the QUIC packet and fill all remaining space in the datagram. By processing the updateable queue first, the implementation ensures that updates for all streams — such as flow control limits — are sent as soon as possible.

4.4.5 Packet Encryption

Applying and removing packet protection is delegated to CryptoSeal class, as described in section 3.8. The CryptoSeal class implements only the logic independent of the specific AEAD cipher used. The steps which are specific to each AEAD cipher, such as the actual in-place encryption and decryption and calculating the header protection mask, are delegeated to an implementation of CryptoSealAlgorithm abstract class. Each supported AEAD cipher has its own CryptoSealAlgorithm implementation.

The interface exposed by CryptoSeal to the ManagedQuicConnection consists of following methods:

`void ProtectPacket(Span<byte> packet, int pnOffset, long pn)`
: Applies packet payload protection and writes AEAD integrity tag at the end of the packet.

`void ProtectHeader(Span<byte> packet, int pnOffset)`
: Applies header protection.

`void UnprotectHeader(Span<byte> packet, int pnOffset)`
: Removes header protection.

`bool UnprotectPacket(Span<byte> packet, int pnOffset, long expectedPn)`
: Attempts to remove the payload protection, returns **true** on success.

The meaning of the method arguments has been left out for brevity, but their purpose should be evident from their usage in source code.

4.4.6 Incoming QUIC Packet Processing

The majority of the ManagedQuicConnection implementation is focused on processing QUIC packets and the QUIC frames they contain. Processing of a QUIC packet can end with one of three possible results which are represented by the ProcessPacketResult enum:

- Ok: Packet was processed without errors.
- DropPacket: Packet should be discarded without informing the peer.
- Error: Received packet violates the protocol and the connection should be closed with an error code.

The process of receiving the UDP datagram with QUIC packets begins in the `ManagedQuicConnection.ReceiveData` method. The individual QUIC packets are processed independently one after another using the following steps:

1. detect the packet type,
2. remove packet protection,
3. parse and validate the packet header fields,
4. check if the packet with same packet number has already been received,
5. register the packet for future acknowledgment, and
6. parse and process all contained QUIC frames.

Because the parsed QUIC frames are represented using **`ref struct`**s (as explained in section 3.5.1), QUIC frames contained in the QUIC packet are parsed and immediately processed one by one, independently of the other QUIC frames. The code processing the individual frame types is organized into separate methods — one for each frame type — for better maintainability.

4.4.7 Generating Outgoing QUIC Packets

The logic which generates outgoing UDP datagrams starts in the `ManagedQuicConnection.SendData` method. When generating outgoing packets, the implementation must first determine whether it has any data to send and, if so, in which QUIC packet type it should be sent. This logic is implemented in the `ManagedQuicConnection.GetWriteLevel` method. Once the packet type to be sent is known, the generation of the actual QUIC packet consists of the following steps:

1. determine the maximum size of the packet that can be sent,
2. compose the packet header,
3. write QUIC frames into the packet payload up to the available packet size,
4. add padding to the packet if necessary[3],
5. apply packet protection, and
6. add the packet to be tracked by the `RecoveryController`.

The order in which the QUIC frames are generated is based on the frames' relative importance to make sure that the important packets fit into the packet and are not unnecessarily postponed. Most importantly, `ACK` frames are written first to avoid delaying acknowledgments, and `STREAM` frames are written last to use up the remainder of the available space.

[3]Header protection (described in section 2.8.2) uses bytes 5 to 20 from the packet payload as a sample for generating the header protection mask. The packet payload includes the packet number, QUIC frames and 16 B AEAD integrity tag. This implies that packet number and QUIC frames together must be at least 4 B long.

4.5 TLS Integration

The TLS handshake related logic of the `ManagedQuicConnection` class is delegated to an implementation of `ITls` interface. The `ITls` interface defines methods needed by `ManagedQuicConnection`. These methods include:

`bool TryAdvanceHandshake()`
: Advances the TLS handshake by calling methods on the `ManagedQuicConnection` (listed later).

`void OnHandshakeDataReceived(EncryptionLevel level, ReadOnlySpan<byte> data)`
: Provides the TLS implementation with data received from the peer via `CRYPTO` frames.

`TlsCipherSuite GetNegotiatedCipher()`
: Gets the identifier of the AEAD cipher that was negotiated during the TLS handshake.

`TransportParameters GetPeerTransportParameters()`
: Returns a `TransportParameters` instance which contains the transport parameters set by the peer for this connection.

`bool IsHandshakeComplete { get; }`
: Returns true if the TLS handshake is considered complete by the TLS implementation.

Implementations of `ITls` are expected to maintain a reference to the `ManagedQuicConnection` and call following functions from the `TryAdvanceHandshake` method when appropriate:

`void SetEncryptionSecrets(EncryptionLevel level, ReadOnlySpan<byte> read, ReadOnlySpan<byte> write)`
: Provides the `ManagedQuicConnection` with read and write secrets negotiated for the specified encryption level. The encryption level refers to one of the four encrypted QUIC packet: `Initial`, `Handshake`, `Application` (1-RTT) and `EarlyData` (0-RTT).

`void AddHandshakeData(EncryptionLevel level, ReadOnlySpan<byte> data)`
: Adds TLS handshake data to be sent to the peer via `CRYPTO` frames. The encryption level specifies which QUIC packet should be used to send the `CRYPTO` frame. This method can be called multiple times.

`void FlushHandshakeData()`
: Called after `AddHandshakeData` to inform `ManagedQuicConnection` that the handshake data should be sent.

`void SendTlsAlert(EncryptionLevel level, int alertCode)`
: When TLS implementation encounters an error, this function is used to provide the TLS alert code which is then used to construct a `CONNECTION_CLOSE` frame for terminating the connection.

There are two `ITls` interface implementations:

- OpenSslTls: Backed by modified *OpenSSL* with QUIC-enabling API maintained by Akamai [30]. This implementation can be used to interoperate with other QUIC implementations and is used by the QuicImplementationProviders.Managed provider.

- MockTls: Intended to be used when runing functional tests in the continuous integration environment where the modified *OpenSSL* library is not available. This implementation cannot be used to interoperate with other QUIC implementations. MockTls is used by the QuicImplementationProviders.ManagedMockTls provider.

The integration with TLS uses the abstract factory pattern to create new instances of ITls when needed. The ManagedQuicImplementationProvider instance keeps a reference to QuicTlsProvider which is used to create new ITls instances for ManagedQuicConnections. The concrete factory classes are realized by the OpenSslQuicTlsProvider and MockQuicTlsProvider classes.

Following subsections describe the two ITls implementations in greater detail.

4.5.1 OpenSslTls Implementation

The OpenSslTls class is a managed wrapper around the SSL class from *OpenSSL* library written in C. As per the established practice in the .NET runtime repository, the managed QUIC implementation separates the definitions of the extern methods from the *OpenSSL* library into a separate Interop.OpenSslQuic class.

The primary classes used from the *OpenSSL* library are SSL_CTX which is a top level object maintaining global SSL/TLS configuration, and SSL which represents one SSL/TLS session. The OpenSslTls implementation uses one global SSL_CTX object, and one SSL object for each OpenSslTls instance.

The main part of the QUIC-enabling API exposed by the *OpenSSL* library consists of SSL_set_quic_method function which registers callbacks from the TLS handshake state machine. Listing 4.3 lists the definition of the *OpenSSL* QUIC callback functions (written in C).

Listing 4.3: Callback definitions in OpenSSL library source code

```c
struct ssl_quic_method_st {
    int (*set_encryption_secrets)(SSL *ssl, OSSL_ENCRYPTION_LEVEL level,
                                  const uint8_t *read_secret,
                                  const uint8_t *write_secret,
                                  size_t secret_len);
    int (*add_handshake_data)(SSL *ssl, OSSL_ENCRYPTION_LEVEL level,
                              const uint8_t *data, size_t len);
    int (*flush_flight)(SSL *ssl);
    int (*send_alert)(SSL *ssl, enum OSSL_ENCRYPTION_LEVEL level,
                      uint8_t alert);
};
typedef struct ssl_quic_method_st SSL_QUIC_METHOD;

int SSL_CTX_set_quic_method(SSL_CTX *ctx, const SSL_QUIC_METHOD *quic_method);
```

Listing 4.4 lists the definition of the QuicMethodCallbacks defined by the interop layer of the managed QUIC implementation which mirrors the ssl_quic_method_st on the managed .NET side. The QuicMethodCallbacks class uses the C# function pointers feature introduced in .NET 5 to represent unmanaged pointers to C# methods [4].

Listing 4.4: C# mirror of the ssl_quic_method_st C struct. The comments above the fields list C-like definition of the function pointer type.

```
[DllImport(Libraries.Ssl, EntryPoint = "SSL_set_quic_method")]
internal static extern unsafe int SslSetQuicMethod(IntPtr ssl,
    QuicMethodCallbacks* methods);

[StructLayout(LayoutKind.Sequential)]
internal unsafe struct QuicMethodCallbacks
{
    // int (*)(IntPtr ssl, OpenSslEncryptionLevel level, byte* readSecret,
    //          byte* writeSecret, UIntPtr secretLen)
    internal delegate* unmanaged[Cdecl]<IntPtr, OpenSslEncryptionLevel,
        byte*, byte*, UIntPtr, int> SetEncryptionSecrets;

    // int (*)(IntPtr ssl, OpenSslEncryptionLevel level, byte* data,
    //          UIntPtr len)
    internal delegate* unmanaged[Cdecl]<IntPtr, OpenSslEncryptionLevel,
        byte*, UIntPtr, int> AddHandshakeData;

    // int (*)(IntPtr ssl)
    internal delegate* unmanaged[Cdecl]<IntPtr, int> FlushFlight;

    // int (*)(IntPtr ssl, OpenSslEncryptionLevel level, byte alert)
    internal delegate* unmanaged[Cdecl]<IntPtr, OpenSslEncryptionLevel, byte,
        int> SendAlert;
}
```

The pointer to QuicMethodCallbacks structure passed to the SSL_set_quic_method function must be valid throughout the lifetime of the SSL_CTX object. Therefore, the OpenSslTls class allocates the QuicMethodCallbacks instance into an unmanaged memory region using the Marshal.AllocHGlobal() method and stores the resulting pointer in a **private static** field.

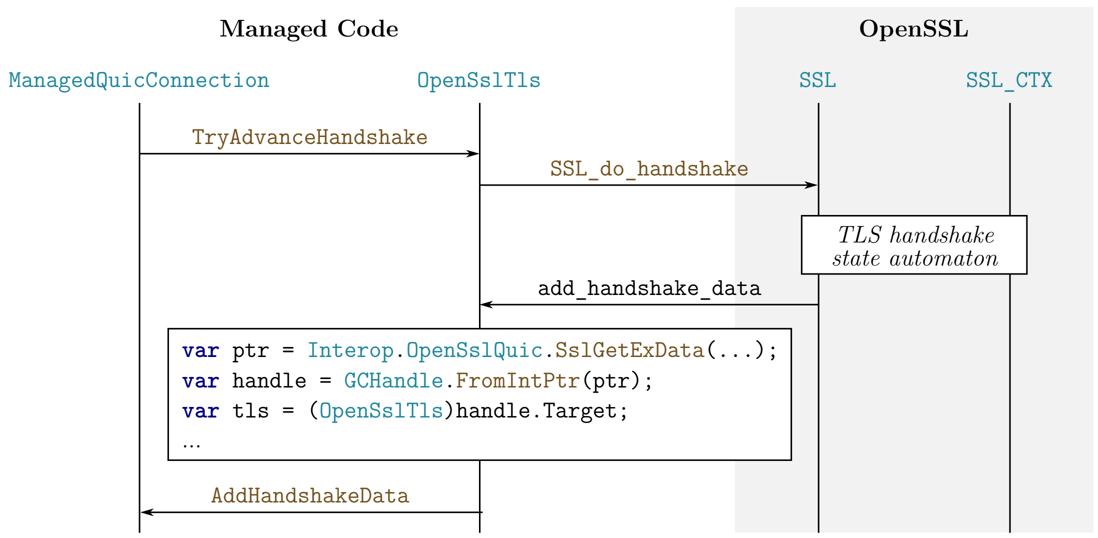
In order to be able to create unmanaged function pointers to C# methods, the actual methods must be annotated using the UnmanagedCallersOnlyAttribute specifying that the caller will use the cdecl calling convention.

The last piece of the ***OpenSSL*** callback integration is calling apropriate methods on the ManagedQuicConnection class from the callbacks. Because the callbacks are converted to unmanaged function pointers, they must be **static** methods. Their implementation must, therefore, translate the SSL* pointer to the original OpenSslTls instance. This is done by allocating a GCHandle for the OpenSslTls instance and storing the unmanaged pointer to the handle as user data inside the SSL instance.

Figure 4.4 illustrates the process of calling the AddHandshakeData callback

[4]More information about C# function pointers and other native code interop improvements done in .NET 5 can be found on Microsoft dev blog [39].

as an example. The ManagedQuicConnection instance tries to advance the TLS handshake by calling TryAdvanceHandshake which in turn calls the SSL_do_handshake native method in the *OpenSSL* library. This invokes the next step in the internal state automaton and as a result, the *OpenSSL* calls the static OpenSslTls.AddHandshakeData method which was registered as the add_handshake_data callback. This function retrieves the GCHandle from the SSL instance passed in the callback and invokes the AddHandshakeData method on the original ManagedQuicConnection instance.



```
var ptr = Interop.OpenSslQuic.SslGetExData(...);
var handle = GCHandle.FromIntPtr(ptr);
var tls = (OpenSslTls)handle.Target;
...
```

Figure 4.4: Callback integration with *OpenSSL* library

4.5.2 MockTls Implementation

The MockTls implementation is used for running tests without the dependency on the *OpenSSL* library and is not intended for use in production environment. The implementation imitates the OpenSslTls behavior during a successful TLS handshake and exchanges randomly generated secrets.

4.6 ManagedQuicListener Implementation

The ManagedQuicListener class is only a simple wrapper around the QuicServerSocketContext instance and a Channel<ManagedQuicConnection> of newly accepted connections. The QuicServerSocketContext implementation observes changes in the ManagedQuicConnection.ConnectionState property and inserts the newly established connections into the Channel. The application later retrieves these connections using the ManagedQuicListener.AcceptConnectionAsync() method.

When the ManagedQuicListener is closed, it is necessary to keep the currently established connections alive. Disposing the ManagedQuicListener, therefore, means only that the QuicServerSocketContext instance will not accept new connections. The QuicServerSocketContext class tracks active connections and is disposed when the last ManagedQuicConnection is closed.

4.7 ManagedQuicStream Implementation

As described in section 3.6, the behavior of sending and receiving parts of a QUIC stream is implemented by the `ReceiveStream` and `SendStream` classes. The `ManagedQuicStream` class only checks the validity of arguments passed to the public API and informs the `ManagedQuicConnection` about the amount of data read/written to update flow control limits.

4.7.1 ReceiveStream Implementation

Buffering and control flow considerations for the `ReceiveStream` class have been analyzed in detail in section 3.6.1. The `ReceiveStream` class exposes the following interface to be used by the `ManagedQuicStream` class from the application thread:

`long?  Error { get; }`
: If the sender aborted the stream, this property contains the application-level error code to be reported to the application.

`void RequestAbort(long errorCode)`
: Requests that the stream is aborted by the sender. Results in sending a `STOP_SENDING` frame with the specified application-level error code.

`int Deliver(Span<byte> destination)`
: Copies application data from the stream into the provided `Span<byte>`. Blocks until at least some data are available.

`ValueTask<int> DeliverAsync(Memory<byte> destination, CancellationToken)`
: Asynchronous version of the `Deliver` method.

From the internal background processing thread, the main methods called from the `ManagedQuicConnection` are:

`void OnResetStream(long errorCode)`
: Called when `RESET_STREAM` frame for this stream was received.

`void Receive(long offset, ReadOnlySpan<byte> data, bool fin)`
: Called when `STREAM` frame has been received.

4.7.2 SendStream Implementation

Buffering and control flow considerations for the `SendStream` class have been analyzed in detail in section 3.6.2. The `SendStream` class exposes the following interface to be used by the `ManagedQuicStream` class from the application thread:

`int Enqueue(ReadOnlySpan<byte> data)`
: Adds the provided application data into the stream. If the internal buffering capacity is full, this method blocks until the data can be buffered.

`ValueTask<int> EnqueueAsync(ReadOnlyMemory<byte> data)`
: Asynchronous version of the `Enqueue` method.

`void MarkEndOfData()`
: Marks the stream as finished. No more data can be written into the stream after calling this function. The `FIN` bit will be set in the appropriate `STREAM` frame.

`void RequestAbort(long errorCode)`
: Requests that the stream be aborted with specified errorCode. This method is also called when `STOP_SENDING` frame was received for this stream.

From the internal background processing thread, the main methods called from the `ManagedQuicConnection` are:

`(long offset, long count) GetNextSendableRange()`
: Returns the next consecutive range of data that is in *pending* state (see section 3.6.2). The next data to be sent in STREAM frame will be from this range.

`void CheckOut(Span<byte> destination)`
: Copies the data from the range returned from `GetNextSendableRange` into the destination buffer. The copied data is transitoned into *in-flight* state.

`void OnLost(long offset, long count)`
: Marks the range of data as lost, transitioning them back into *pending* state.

`void OnAck(long offset, long count, bool fin)`
: Marks the range of data as *acknowleged*. Allowing the underlying buffers to be reused.

4.7.3 Tracing and Diagnostics

The managed QUIC implementation can produce verbose logs (called traces) to provide an insight into the implementation's behavior. By default, no trace is generated. Generating a trace can be enabled by defining the `DOTNETQUIC_TRACE` to one of the following values:

`console`
: Produces logs to the console output. This type of tracing produces a large amount of console logs, which are hard to reason about. However, the console logs are the fastest way of checking if any packets are sent/received or for diagnosing refused connections.

`qlog`
: Produces traces which can be visualized by the *qvis* [36] tool. The *qvis* tool provides deep insight into the connection, such as the timeline of sent/received packets, bytes in flight, size of the congestion window, and latency throughout the lifetime of the connection. The traces are written into the current working directory into a separate file for each connection. Note that for very busy connections, the trace files can grow into hundreds of megabytes in a few seconds.

Both types of tracing incur a noticeable overhead and should not be used in the production environment.

4.8 Tests Implementation

The QUIC implementation is covered by an extensive suite of unit tests and functional tests. The unit tests are located in the `System.Net.Quic/tests/UnitTests` directory and focus on the correctness of the individual parts of the managed QUIC implementation. The unit tests also check that the QUIC packets sent by the implementation conform to the QUIC protocol specification.

The functional tests are located in the `System.Net.Quic/tests/FunctionalTests` directory and test high-level functionality like being able to send and receive data, and that the public API behavior conforms to the public API specification.

4.8.1 Unit Tests

The most important part of the unit tests are tests that inspect the contents of the individual QUIC packets sent between the server and client. In order to simplify writing these tests, the `System.Net.Quic.Unit.Tests` project implements a testing harness, which provides helper methods for inspecting and modifying QUIC packets, as discussed in section 3.10.1. The test harness functionality is provided as a `ManualTransmissionQuicTestBase` class intended to be used as a base class for unit test classes. This class provides:

- prepared connected `ManagedQuicConnection` instances;
- automatic logging of sent QUIC packets to the test output;
- manual time stepping to make tests deterministic; and
- interception callbacks for inspecting and possibly modifying QUIC packets in-flight;

Because the `ManagedQuicConnection` implementation uses **`ref struct`**s to represent individual QUIC frames. The testing harness must duplicate the types for QUIC frames and QUIC packets as classes. The classes used to represent QUIC packets and QUIC frames in test harness derive from `FrameBase` and `PacketBase` abstract classes, which provide a common type for aggregating QUIC packets and QUIC frames in collections.

The classes representing QUIC frames and QUIC packets are mutable, which allows modification by the testing code to elicit a particular error response. Figure 4.5 illustrates the process. First, the `ManagedQuicConnection` produces the UDP datagram containing the QUIC packets to be sent. The testing harness then parses the datagram into mutable representation. The parsed packets are then passed into the inspection callback provided by the test method, which can perform any assertions and modifications. The QUIC packet is then serialized back into binary representation and provided to the receiver `ManagedQuicConnection` instance.

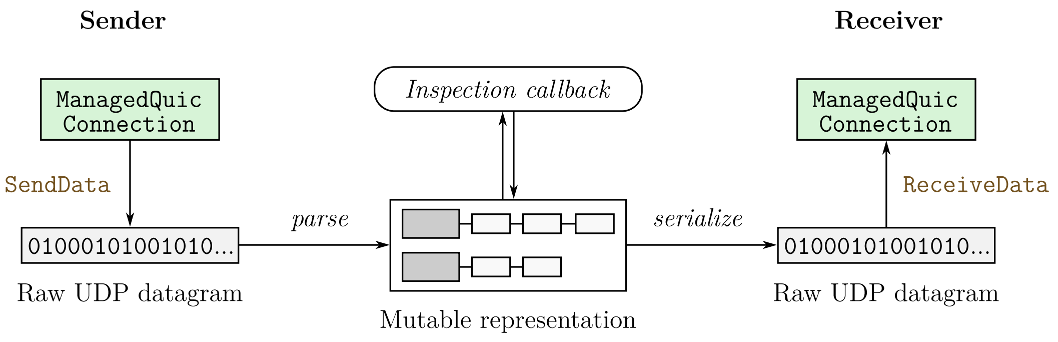


Figure 4.5: Intercepting a QUIC packet by the testing harness

4.8.2 Functional Tests

The `System.Net.Quic` project contains a large suite of functional tests against the public QUIC API. The tests are structured in a way that allows them to be run against all QUIC implementation providers in the library. Also, the `QuicStream` class is tested using *stream conformance tests* — a suite of tests run for each implementation of the `Stream` class to ensure consistent behavior of all implementations.

In order to allow running tests against different implementation providers, the test methods are defined in `QuicListenerTests<T>`, `QuicConnectionTests<T>` and `QuicStreamTests<T>` generic classes, which share a common base class `QuicTestBase<T>`. The type argument for all four of these classes is constrained to be an implementation of `IQuicImplProviderFactory` interface which allows creating `QuicListener` and `QuicConnection` instances with desired implementation provider.

The ***xUnit*** framework does not run tests from generic classes by itself. Instead, the generic test suite must be "instantiated" by declaring a non-generic class deriving from the generic type. For example, `QuicStreamTests_ManagedProvider` derives from the `QuicStreamTests<ManagedProviderFactory>` class which makes ***xUnit*** run the tests from `QuicStreamTests<T>` class against the `QuicImplementationProviders.Managed` provider. These specific test classes are also annotated by `ConditionalClassAttribute` which instructs ***xUnit*** to run the tests only if the given provider is supported in the current environment.

A similar mechanism is used to run stream conformance tests against all QUIC implementation providers.

5. User Documentation

This section provides guidance on how to obtain the build of the .NET runtime with managed QUIC implementation, how to install it, and how to use the code to develop other applications.

5.1 Getting Started

This thesis provides a branch of the .NET runtime codebase with managed QUIC implementation. Since our branch contains changes only in the `System.Net.Quic.dll`, the easiest way of composing a fully working .NET distribution is obtaining a full SDK installation of the latest master development version of .NET 6 and replacing the `System.Net.Quic.dll`. This section explains how to do this without affecting other .NET SDK installations present on the machine.

The managed implementation depends on a particular ***OpenSSL*** version to interoperate with other QUIC implementations. This section also explains how to deploy a locally built ***OpenSSL*** to be automatically used by user code.

The complete setup process consists of the following steps, which we explained in greater detail in the subsequent subsections:

1. Build the `System.Net.Quic.dll` library from our branch of the .NET runtime sources.
2. Compose a new local .NET 6 SDK installation with locally built `System.Net.Quic.dll`.
3. *(optional)* Compile a QUIC-supporting ***OpenSSL*** from source and deploy it.
4. Configure the development environment to use the new .NET installation when compiling and running user applications.

Because the setup process is complex and compiling the necessary binaries requires a lot of prerequisite tools to be installed on the machine, this thesis attachments include binaries and .NET 6 SDK built and prepared according the instructions in this section. The files can be found in the `bin/win-x64/` directory for Windows and `bin/linux-x64/` directory for Linux.

5.1.1 Building the System.Net.Quic Library from Source

The source code for the .NET runtime with managed QUIC implementation is part of this thesis' attachments in the `src/dotnet-runtime/` directory. The latest version of the source code can also be found on the thesis author's GitHub [40]. For the remainder of this section, all paths will be relative to the .NET runtime repository directory.

The .NET runtime repository contains a descriptive guide on how to build the sources. The necessary prerequisites are listed in files inside the `docs/workflow/requirements/` directory, separately for each operating system. Once all necessary prerequisities are installed, the entire .NET runtime can be built using the `build.cmd` batch file (on Windows) or `build.sh` script (on Linux). The arguments are the same for both operating systems.

```
> ./build.cmd -subset clr+libs -configuration release
```

The above command will build the Common Language Runtime (CLR) and all libraries in Release configuration. The artifacts are available in the `artifacts/bin/System.Net.Quic` directory. The important artifacts from this directory are:

- `ref/net6.0-Release/System.Net.Quic.dll`: The so-called *reference assembly* [41] that specifies the public API of the library.
- `net6.0-{OS}-Release/System.Net.Quic.dll`: Where `{OS}` is the identifier for the operating system running on the machine. This is the .NET assembly with the actual QUIC implementation.

5.1.2 Creating a Local Installation of .NET

Now we need to download the latest .NET 6 SDK and patch it with the locally built `System.Net.Quic.dll`. A zip archive containing the SDK can be downloaded from a link listed in the official SDK installer GitHub repository [42]. Download the "Master (6.0.x Runtime)" build for your platform and extract it to a convenient location. In the remainder of this guide, the directory containing the extracted contents will be referred to as `DOTNET_ROOT`.

The `System.Net.Quic.dll` produced in the previous subsection must be copied to appropriate locations in the `DOTNET_ROOT`. The reference `System.Net.Quic.dll` assembly should be copied over the existing one in the `DOTNET_ROOT/packs/Microsoft.NETCore.App.Ref/6.0.0-{version}/ref/net6.0/` directory, and the implementation assembly to should be copied to the `DOTNET_ROOT/shared/Microsoft.NETCore.App/6.0.0-{version}/` directory, overwriting the existing files.

5.1.3 Building the OpenSSL Library

The implementation requires a QUIC-supporting *OpenSSL* library build from a development branch maintained by Akamai. Because this branch does not offer official builds for downloading, the library must be compiled from source. Alternatively, a built version of the library is attached in the `bin/{platform}/openssl/` directory.

The appropriate source codes can be found in the `extern/openssl` directory in the thesis attachments. The source code is also available online on Akamai's GitHub [30]. The implementation has been developed and tested with the `OpenSSL_1_1_1g-quic` branch of the code, but other QUIC-enabled branches of *OpenSSL* version 1.1.1 should work as well.

Before building the *OpenSSL* library form source, check the `NOTES.{OS}` file in the repository and make sure all prerequisites are installed on the machine. After that, the *OpenSSL* library can be built by running the following command inside the repository. Note that for Windows OS, you must run these commands using the *x64 Native Tools Command Prompt for VS* in order to have the necessary tools in `PATH`.

```
# Windows
> perl Configure VC-WIN64A
> nmake

# Linux
> ./config
> make
```

This will produce the ***libcrypto*** and ***libssl*** libraries in the ***OpenSSL*** repository root. On windows, these libraries are named `libcrypto-1_1-x64.dll` and `libssl-1_1-x64.dll`. These libraries are loaded by the managed QUIC implementation during runtime and, therefore, must be present in a location where the OS loader can find them. This can be achieved by putting the libraries in any of the following locations:

- Next to the compiled program executable.
- A directory listed in the `PATH` environment variable
- *(preferred)* next to the `System.Net.Quic.dll` library in the .NET installation directory, i.e., `DOTNET_ROOT/shared/Microsoft.NETCore.App/6.0.0-{version}/`.

Note that if there is already a different version of ***OpenSSL*** installed on the system, it is necessary to ensure that the system loads the correct ***OpenSSL*** version. This is different for each operating system:

- On Windows, this can be ensured by placing the DLL files in the same directory as the program executable. The entire library search process, including the order of directories searched, is described in Windows documentation [43] in detail.
- On Linux, this can be achieved by defining the `LD_LIBRARY_PATH` environment variable to the directory containing the ***OpenSSL*** libraries. Additional information about loading of dynamic libraries on Linux can be found in the manual pages for `ld.so`. These manual pages are also available online [44].

5.1.4 Configuring the Development Environment

Lastly, we need to configure the environment variables so that the .NET SDK installation created in the previous step is used when building the user code. For this, the following environment variables need to be defined correctly.

`DOTNET_ROOT`
Path to the .NET installation directory. This instructs the build process to use the SDK installed in this directory. Use the path to the local .NET 6 SDK installation created in section 5.1.2.

DOTNET_MULTILEVEL_LOOKUP
: Set this to “0”. This instructs the build process not to look for SDK installation in other places than `DOTNET_ROOT`.

PATH
: Prepend the `DOTNET_ROOT` directory to the beginning of the `PATH` variable to make sure the `dotnet` executable from the `DOTNET_ROOT` is used over the system-wide installed one.

After configuring the variables, check the output of the `dotnet --info` command. Assuming `DOTNET_ROOT` is `C:\dotnet\\`, then the output should be similar to the Listing 5.1. The list of installed .NET runtimes should contain `Microsoft.NETCore.App` from the local .NET 6 SDK installation prepared in **??** (check the path inside the brackets). Note that the listing contains version numbers of the latest .NET 6 SDK at the time of writing this text, and the SDK installer would be updated since then to a newer version.

Listing 5.1: Output of the `dotnet --info` command in correctly configured environment. The unimportant portions of the output in grey has been left out brevity

```
> dotnet --info
.NET SDK (reflecting any global.json):
 ...

Runtime Environment:
 ...

Host (useful for support):
 ...

.NET SDKs installed:
  6.0.100-alpha.1.20563.2 [C:\dotnet\sdk]

.NET runtimes installed:
  Microsoft.AspNetCore.App 6.0.0-alpha.1.20526.6 [C:\dotnet\shared\...]
  Microsoft.NETCore.App 6.0.0-alpha.1.20560.10 [C:\dotnet\shared\...]
  Microsoft.WindowsDesktop.App 6.0.0-alpha.1.20560.7 [C:\dotnet\shared\...]

To install additional .NET runtimes or SDKs:
  https://aka.ms/dotnet-download
```

With the environment configured as described, .NET applications can be compiled against the .NET 6 SDK either using the `dotnet build` command-line command, or using Visual Studio or any other IDE.

5.1.5 Creating a Sample .NET 6 Project

The last step is creating a new project and configuring it to use .NET 6. This section demonstrates how this can be done using the `dotnet` command-line tool. Assuming the environment variables have been configured as specified in section 5.1.4, you can create a new project using the following commands:

```
> mkdir hello-net6
> cd hello-net6
> dotnet new console
```

These commands will create a `hello-net6/hello-net6.csproj` file. Use of .NET 6 preview requires minor changes to the project file. Namely changing the `TargetFramework` property to `net6.0`. The modified project file contents are listed in Listing 5.2.

Listing 5.2: Project file for .NET 6 console application project.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <RootNamespace>hello_net6</RootNamespace>
  </PropertyGroup>

</Project>
```

Lastly, the NuGet package feed must be configured to use the correct package source for the preview packages for .NET 6. This can be done by creating a `NuGet.Config` file in the project directory with contents as listed in Listing 5.3. The contents can also be copied from the webpage from which the .NET 6 SDK was downloaded [42].

Listing 5.3: NuGet configuration file for .NET 6 projects.

```
<configuration>
  <packageSources>
    <add key="dotnet6"
      value="https://pkgs.dev.azure.com/dnceng/public/_packaging/dotnet6/nuget/v3/index.json" />
  </packageSources>
</configuration>
```

To check that the development environment is configured correctly, the program in Listing 5.4 can be used.

Listing 5.4: C# program for testing the SDK installation.

```
using System;
using System.Diagnostics;
using System.IO;
using System.Net.Quic;

namespace hello_net6._0
{
    class Program
    {
        static void Main(string[] args)
        {
            string assemblyPath = typeof(object).Assembly.Location;
            string assemblyDir = Path.GetDirectoryName(assemblyPath);
            var info = FileVersionInfo.GetVersionInfo(assemblyPath);
            Console.WriteLine($"Hello from .NET {info.ProductVersion}");
            Console.WriteLine($"Runtime location: {assemblyDir}");
            Console.WriteLine($"QUIC: {QuicImplementationProviders.Default}");
        }
    }
}
```

Running the program should produce output similar to the following:

```
> dotnet run
Hello from .NET 6.0.0-alpha.1.20560.10+72b7d236ad634c2280c73499ebfc2b594995ec06
Runtime location: C:\dotnet\shared\Microsoft.NETCore.App\6.0.0-alpha.1.20560.10
QUIC: System.Net.Quic.Implementations.Managed.ManagedQuicImplementationProvider
```

If the output lists a different runtime location, verify that the environment variables have been set correctly. If the QUIC provider is different, it means that the `System.Net.Quic.dll` with our managed QUIC implementation was not copied to the correct directory.

5.1.6 Deploying .NET Applications

The applications built against the preview .NET 6 SDK will run only if the environment is configured according to section 5.1.4. In order to run the applications outside the configured environment, it is necessary to build them as *self-contained* [45]. Self-contained builds of .NET applications contain a copy of the .NET runtime and other necessary binaries. Self-contained applications can be built using following command:

```
> dotnet publish --self-contained --runtime <RID>
```

Where `<RID>` is the runtime identifier for which to publish. Commonly used values are `win-x64` and `linux-x64`. Full list of supported runtime identifiers can be found in the official .NET documentation [46].

Unfortunately, the packaged application contains the unmodified `System.Net.Quic.dll` without managed QUIC support. Therefore, the last step is manually overwriting the `System.Net.Quic.dll` with the one locally built from the .NET runtime sources in section 5.1.1. The modified *OpenSSL* libraries also need to be copied over to the application directory.

5.2 Simple Echo Server using QUIC

This section is a walkthrough on how to use QUIC in .NET. In this section, we will create a trivial echo server. When a new connection is established, clients will open a single bidirectional stream and send arbitrary data over it. The server will then echo the data back to the client using the same stream. For simplicity, we will use a single .NET program to represent both client and server and conduct the connection over the loopback network interface. We will also omit error checking from this example for brevity. The complete source code for this example can be found in the `src/supplementary/samples/Echo/` directory in the thesis attachments.

5.2.1 Echo Server Implementation

The use of TLS 1.3 for encryption is mandatory for QUIC. This requires providing an X.509 certificate on the server's side. The current QUIC API requires that the certificate and the private key be saved in separate files in the PEM format. Certificate files that can be used in this example are provided in the attachments. The public certificate file is at `certs/cert.crt` and the private key is at `certs/cert.key`. Alternatively, a new certificate can be created using the `openssl` command-line utility using the following commands.

```
> openssl req -x509 -newkey rsa -keyout key.pem -out cert.pem -days 365 -nodes
```

In order to accept incoming QUIC connections, we need to create an instance of `QuicListener`. Listing 5.5 shows how the `QuicListener` can be created and provided with:

- listening endpoint;
- identifier of the application-layer protocol to be used[1], even though we are not implementing any standard protocol, we still need to provide one; for our example, we chose to use `"echo"` as the ALPN identifier; and
- paths to the X.509 certificate and private key file.

The `QuicListener` class implements `IDisposable` and, therefore, we can also use the **`using`** statement to make sure the `QuicListener` is closed when the method returns.

Listing 5.5: Creating and starting a new `QuicListener` for the echo server

```
public static async Task<int> RunServer(IPEndPoint listenEp,
    string certificateFile, string privateKeyFile, CancellationToken token)
{
    using QuicListener listener = new QuicListener(new QuicListenerOptions
    {
        ListenEndPoint = listenEp,
        CertificateFilePath = certificateFile,
        PrivateKeyFilePath = privateKeyFile,
```

[1]QUIC is not intended to be used standalone, but as a transport protocol for other application-layer protocols. Since servers can support multiple versions of the application protocol, QUIC uses the ALPN extension to TLS to negotiate the application-layer protocol as part of the QUIC connection handshake.

```
        ServerAuthenticationOptions = new SslServerAuthenticationOptions
        {
            ApplicationProtocols = new List<SslApplicationProtocol>
            {
                new SslApplicationProtocol("echo")
            }
        }
    });

    // ...
}
```

We can then use the AcceptConnectionAsync method to wait for new incoming connections asynchronously. Listing 5.6 shows how to accept new connections and process them asynchronously in a separate Task so that multiple connections can be served concurrently.

Listing 5.6: Accepting new connections on QuicListener

```
    // QuicListener must be started before accepting connections.
    listener.Start();

    // tasks that need to be awaited when trying to exit gracefully
    List<Task> tasks = new List<Task>();

    try
    {
        QuicConnection conn;
        while ((conn = await listener.AcceptConnectionAsync(token)) != null)
        {
            // copy the connection into a variable with narrower scope which
            // can be safely captured inside the lambda function
            QuicConnection captured = conn;
            var task = Task.Run(
                () => HandleServerConnection(captured, token));
            tasks.Add(task);
        }
    }
    finally
    {
        // wait until all connections are closed
        await Task.WhenAll(tasks);
    }
```

Listing 5.7 Shows the implementation of HandleServerConnection which does the actual echoing of the incoming data. The QuicStream is accepted using the AcceptStreamAsync method on the QuicConnection class and, because it is a bidirectional stream, we can use it to send the data back to the client. Lastly, once all data is sent, we gracefully close the connection using the CloseAsync method.

Listing 5.7: Echo server reading and writing data to `QuicStream`

```
public static async Task HandleServerConnection(QuicConnection connection,
    CancellationToken token)
{
    try
    {
        QuicStream stream = await connection.AcceptStreamAsync(token);

        int read;
        byte[] buffer = new byte[4 * 1024];
        while ((read = await stream.ReadAsync(buffer, token)) > 0)
        {
            await stream.WriteAsync(buffer, 0, read, token);
            await stream.FlushAsync(token);
        }
    }
    finally
    {
        // gracefully close the connection with 0 error code
        await connection.CloseAsync(0);
    }
}
```

5.2.2 Echo Client Implementation

The client implementation is more straightforward than that of the server. Listing 5.8 shows how to create a client `QuicConnection` using the using the server endpoint address and the ALPN identifier. The connection can be then established by calling the `ConnectAsync` method.

Listing 5.8: Creating a client `QuicConnection` instance

```
public static async Task<int> RunClient(IPEndPoint serverEp,
    CancellationToken token)
{
    using var client = new QuicConnection(new QuicClientConnectionOptions
    {
        RemoteEndPoint = serverEp,
        ClientAuthenticationOptions = new SslClientAuthenticationOptions
        {
            ApplicationProtocols = new List<SslApplicationProtocol>
            {
                new SslApplicationProtocol("echo")
            }
        }
    });

    await client.ConnectAsync(token);

    // ...
}
```

Once the connection is established, the client opens a bidirectional `QuicStream` using the `OpenBidirectionalStream` method. The `QuicStream` can be used like any other `Stream` instance. Listing 5.9 shows the rest of the echo client implementation.

Listing 5.9: Sending standard input via `QuicStream`

```
try
{
    await using QuicStream stream = client.OpenBidirectionalStream();

    // spawn a reader task to not let server be flow-control blocked
    _ = Task.Run(async () =>
    {
        byte[] arr = new byte[4 * 1024];
        int read;
        while ((read = await stream.ReadAsync(arr, token)) > 0)
        {
            string s = Encoding.ASCII.GetString(arr, 0, read);
            Console.WriteLine($"Received: {s}");
        }
    });

    string line;
    while ((line = Console.ReadLine()) != null)
    {
        // convert into ASCII byte array before sending
        byte[] bytes = Encoding.ASCII.GetBytes(line);
        await stream.WriteAsync(bytes, token);
        // flush the stream to send the data immediately
        await stream.FlushAsync();
    }

    // once all stdin is written, close the stream
    stream.Shutdown();

    // wait until the server receives all data
    await stream.ShutdownWriteCompleted(token);
}
finally
{
    // gracefully close the connection with 0 error code
    await client.CloseAsync(0, token);
}
```

5.2.3 A More Complex Example Application

In this thesis, we developed a more complex version of the echo server example from the previous section for benchmarking purposes. The source code for this benchmarking application can be found in the `src/supplementary/benchmark/ThroughputTests/` directory of the thesis attachments. We will provide more information about the application in section 6.1.3.

5.3 QUIC API Reference

This section describes the API designed by the .NET development team to expose QUIC to other developers. As mentioned in the introduction chapter, the current design is a work-in-progress and is subject to change in the future. All of the mentioned classes are located in the `System.Net.Quic` namespace.

5.3.1 QuicListener Class

The `QuicListener` class is the equivalent of the `TcpListener` for TCP connections. Servers use this class to accept incoming QUIC connections.

`QuicListener(QuicListenerOptions)`
: Constructor.

`IPEndPoint ListenEndPoint { get; }`
: The IP endpoint being listened to for new connection. Read-only.

`ValueTask<QuicConnection> AcceptConnectionAsync(CancellationToken)`
: Accepts a new incoming QUIC Connection.

`void Start()`
: Starts listening.

`void Close()`
: Stops listening and closes the listener. Does not close already accepted connections.

5.3.2 QuicListenerOptions Class

The `QuicListenerOptions` class holds all configuration used to construct new `QuicListener` instances.

`SslServerAuthenticationOptions ServerAuthenticationOptions { get; set; }`
: SSL related options like certificate selection/validation callbacks, and supported protocols for ALPN.

`string CertificateFilePath { get; set; }`
: Path to the X.509 certificate used by the server.

`string CertificateKeyPath { get; set; }`
: Path to the private key for the used X.509 certificate.

`IPEndPoint ListenEndPoint { get; set; }`
: The IP endpoint to listen on.

`int ListenBacklog { get; set; }`
: Number of connection to be held waiting for acceptance by the application. Upon reaching this limit, further connections will be refused.

`long MaxBidirectionalStreams { get; set; }`
: Limit on the number of bidirectional streams the client can open in an accepted connection.

`long MaxUnidirectionalStreams { get; set; }`
: Limit on the number of unidirectional streams the client can open in an accepted connection.

`TimeSpan IdleTimeout { get; set; }`
: The period of inactivity after which the connection will be closed via idle timeout.

5.3.3 QuicConnection Class

The `QuicConnection` class represents the QUIC connection itself. Clients open new connections by creating a new instance of this class and calling the `ConnectAsync` method. Servers receive new connections using the `QuicListener` class.

`QuicConnection(QuicClientConnectionOptions)`
: Constructor. The newly created instance must be explicitly connected using the `ConnectAsync` method.

`bool Connected { get; }`
: Indicates whether the `QuicConnection` is connected (the handshake has completed).

`IPEndPoint LocalEndPoint { get; }`
: Local IP endpoint of the connection.

`IPEndPoint RemoteEndPoint { get; }`
: Remote IP endpoint of the connection.

`ValueTask ConnectAsync(CancellationToken)`
: Connects to the remote endpoint.

`QuicStream OpenUnidirectionalStream()`
: Opens a new unidirectional stream. Throws a `QuicException` if the stream cannot be opened.

`QuicStream OpenBidirectionalStream()`
: Opens a new bidirectional stream. Throws a `QuicException` if the stream cannot be opened.

`ValueTask<QuicStream> AcceptStreamAsync(CancellationToken)`
: Accepts an incoming stream.

`ValueTask CloseAsync(long, CancellationToken)`
: Closes the connection with the specified given error code and terminates all active streams.

`long GetRemoteAvailableUnidirectionalStreamCount()`
: Gets the maximum number of unidirectional streams that this endpoint can open.

`long GetRemoteAvailableBidirectionalStreamCount()`
: Gets the maximum number of bidirectional streams that this endpoint can open.

5.3.4 QuicClientConnectionOptions

The `QuicClientConnectionOptions` is used by clients to configure new QUIC conections.

`SslClientAuthenticationOptions ClientAuthenticationOptions { get; set; }`
: Client authentication options to use when establishing the connection.

`IPEndPoint LocalEndPoint { get; set; }`
: The local IP endpoint that will be bound to.

`IPEndPoint RemoteEndPoint { get; set; }`
: The IP endpoint to connect to.

`long MaxBidirectionalStreams { get; set; }`
: Limit on the number of bidirectional streams the server can open.

`long MaxUnidirectionalStreams { get; set; }`
: Limit on the number of unidirectional streams the server can open.

`TimeSpan IdleTimeout { get; set; }`
: The period of inactivity after which the connection will be closed via idle timeout.

5.3.5 QuicStream Class

The `QuicStream` class represents a single stream in a QUIC connection and derives from the abstract `Stream` class. The `Stream` class is a bidirectional stream abstraction and since not all QUIC streams are bidirectional, user should check if the specific `QuicStream` instance supports supports the operation by inspecting the `CanRead` and `CanWrite` properties. Invoking write methods on read-only — more specifically, incoming unidirectional — stream will cause an `InvalidOperationException` to be thrown and vice versa.

The list below mentions the members specific for the `QuicStream` class and some important members inherited from the `Stream` class.

`long StreamId { get; }`
: The Stream ID.

`bool CanRead { get; }`
: Returns `true` if the stream supports reading.

`bool CanWrite { get; }`
: Returns `true` if the stream supports reading.

`void AbortRead(long)`
: Aborts the receiving part of the stream with the provided error code.

`void AbortWrite(long)`
: Aborts the sending part of the stream with the provided error code.

`int Read(Span<byte>)`
: Reads the content of the stream into the provided buffer, blocks if no data is available. Returns 0 only when there will be no more data in the stream.

`ValueTask<int> ReadAsync(Memory<byte>, CancellationToken)`
: Reads the content of the stream into provided buffer, blocks until some data is available. Returns 0 only when there will be no more data in the stream.

`void Write(Span<byte>)`
: Writes the content of the provided buffer into the stream, returns when the data have been buffered internally.

`ValueTask WriteAsync(*, CancellationToken)` *(multiple overloads)*
: Multiple overloads of this method offer writing from various types of buffers: `ReadOnlyMemory<byte>`, `ReadOnlySequence<byte>`, and `ReadOnlyMemory<ReadOnlyMemory<byte>>`. The last one can be used to perform *Vectored I/O* [47]. The returned task completes when the provided data have been buffered internally and the buffers can be reused for other purposes.

`ValueTask WriteAsync(*, bool, CancellationToken)` *(multiple overloads)*
: Like the methods above, but also allow specifying that the provided data are the last on the stream and that the stream should be gracefully closed.

`ValueTask ShutdownWriteCompleted(CancellationToken)`
: The returned task completes when the stream shutdown completes. Meaning that acknowledgment from the peer is received.

`ValueTask Shutdown()`
: Gracefully closes the writing direction of the stream, indicating that no more data will be sent.

5.3.6 Exceptions

The QUIC API can throw the following exceptions:

`QuicException`
: Base class for all thrown exceptions, used when a more specific exception is not available

`QuicConnectionAbortedException`
: Thrown when the connection is forcibly closed either by the transport or by the remote endpoint.

`QuicStreamAbortedException`
: Thrown when the stream was aborted by the remote endpoint.

`QuicOperationAbortedException`
: Thrown when the pending operation was aborted by the local endpoint.

6. Evaluation

This chapter evaluates the performance of the managed QUIC implementation developed in this thesis and compares it to the existing *MsQuic*-based implementation. We will also compare both managed and *MsQuic*-based QUIC implementations to `SslStream` which uses a combination of TCP and TLS. In our evaluation, we will focus on two performance characteristics relevant for network communication:

- *Throughput*: The rate at which the application data are sent. High total throughput is essential for servers when handling a large number of parallel connections.

- *Round Trip Time Latency*: The delay between sending a request and receiving a response. This metric is important mainly for clients, where lower latencies mean faster responses from the server. Rather than measuring the average latency, it is more common to measure, e.g., the 99th percentile of latency, which indicates minimum expected latencies in the slowest 1% replies. This metric is vital in the context of webpages and communication with browsers, especially for websites for which web browsers generate a large number (even hundreds) of HTTP requests for a single page access. In such cases, the probability of at least one of the requests being slower than the 99th percentile raises significantly and, therefore, a high value of the 99th percentile would negatively affect the total page load time for a significant number of users [48].

6.1 Evaluation Environment

We have evaluated our implementation in two different environments. Our primary evaluation environment consists of two blade servers on the same LAN network. For short, we will reffer to this environment as the *Linux LAN* environment. The relevant software and hardware parameters of the servers are:

- *CPU*: Intel® Xeon® E3-1270 v6 @ 3.80 GHz (4.20 GHz turbo boost, 4 cores, 2 threads per core)

- *RAM*: 32 GB (2400 MHz)

- *Network Interface Card*: Intel® Ethernet Controller I350 (1 Gbit/s)

- *OS*: Fedora 32

Unfortunately, the two servers are connected only by 1 Gbit/s Ethernet, which implies a theoretical upper limit for TCP's throughput at 117.5 MB/s and similar limit for UDP. In some measurements, we reached values that approached this limit. Therefore, when appropriate, we also evaluated our implementation using a loopback network interface on a desktop workstation PC. For short, we will reffer to this environment as *Win loopback*. The machine has following specifications:

- *CPU*: Intel® Core® i7-8700 @ 3.20 GHz (4.60 GHz turbo boost, 6 cores, 2 threads per core)
- *RAM*: 32 GB (2400 MHz)
- *OS*: Windows 10 version 20H2[1]

In both environments, the testing application was run in a a 64-bit process on preview .NET 6.0.0 host (CoreCLR 6.0.20.55508, CoreFX 6.0.20.255508).

6.1.1 Drawbacks of Measuring Loopback Performance

Measuring over the loopback interface removes any upper limit on bandwidth because the loopback interface is implemented purely in software inside the operating system. It also minimizes the transport latency and guarantees zero packet loss during transport. The software-based implementation of the loopback interface allows optimizations that are not feasible on Ethernet networks. As an example, on Windows operating system, loopback connections can send 64 KiB IP packets — which is the maximum size allowed by the IP protocol — without any link-layer fragmentation. On the other hand, Ethernet connections can carry only up to 1500 B IP datagrams without fragmentation. The greater size of datagrams on the loopback interface reduces overhead when TCP is used for inter-process communication on the same machine.

Both managed and ***MsQuic***-based QUIC implementations send IP datagrams which are at most 1500 B even on loopback connections, because of a hardcoded upper limit on UDP datagram size in both implementations[2]. In order to make the loopback connection measurements more comparable to those done on Ethernet connections, we leveraged the ability to limit the maximum outbound IP packet size to 1500 B on Windows sockets API for all loopback TCP connections in our tests. From C#, this can be achieved using following statement:

```
socket.SetSocketOption(SocketOptionLevel.IP,
  (SocketOptionName)76 /* IP_USER_MTU */, 1500 /* size */);
```

However, the `IP_USER_MTU` option is only available on Windows 10 version 20H2 or newer. On other platforms, the `SetSocketOption` call above would throw an exception. In order to keep cross-platform compatibility, the relevant lines have been commented out in the attached version of the source code. However, they was used to obtain the results presented in this chapter.

[1]Update to Windows 10 20H2 includes significant TCP and UDP performance improvements [49]. Measurements taken on older versions of Windows would yield significantly different results.

[2]QUIC does not impose an upper limit on the UDP datagram size. However, it requires that the UDP datagrams be small enough to avoid fragmentation. The hardcoded limit has been chosen to simplify the implementation and to work well on Ethernet connections. This limit may be removed in future versions of the libraries

6.1.2 Running with MsQuic Support

MsQuic-based QUIC implementation does not work in the default installation of .NET because .NET does not distribute the *MsQuic* binary. Applications that wish to use the *MsQuic*-based QUIC implementation should reference the `System.Net.Experimental.MsQuic` NuGet package [50] which provides the necessary *MsQuic* binary. However, the Windows *MsQuic* binary distributed in the NuGet package requires a Windows Insider Preview build of Windows because it depends on newer *Schannel* version for TLS implementation, which is not available in the mainstream Windows version.

Unfortunately, we cannot install a preview build of Windows on the workstation we will use for running the tests. Fortunately, in order to support Linux, *MsQuic* can also use the same modified *OpenSSL* version for the TLS implementation as our managed QUIC implementation. Even though building *MsQuic* with *OpenSSL* for Windows is not officially supported, we were able to produce such a build with small changes in the *MsQuic* build configuration. Therefore, the *MsQuic* library used in experiments in this chapter has been compiled locally using source code from commit `dc2a6cf0dd12e27` from the official *MsQuic* repository [14] and configured to use the same *OpenSSL* library as the managed QUIC implementation. It should be noted that our custom Windows build of *MsQuic* may show slightly different performance characteristics than the official build for Windows, which uses the *Schannel* library.

On Linux, we could use the above mentioned `System.Net.Experimental.MsQuic` NuGet package, but that would complicate the build process because the package would have to be referenced only on Linux builds. Therefore, we have decided to compile the Linux build of *MsQuic* locally as well. Like the Windows build, the Linux build of *MsQuic* used in our tests was compiled from commit `dc2a6cf0dd12e27` but did not require changes to build configuration, as it uses the modified *OpenSSL* library by default. Built *MsQuic* binaries for both Windows and Linux operating systems which were used in tests can be found in the `bin/{platform}/msquic/` directory of this thesis attachments.

All measurements of the *MsQuic*-based QUIC support for .NET include any overhead introduced by the interop layer bridging the native library API to .NET QUIC API. The measured results, therefore, do not represent raw *MsQuic* library performance, but the performance observed by .NET applications which use the *MsQuic*-based implementation provider which can be validly compared to the performance of our managed QUIC implementation.

6.1.3 Benchmarking Application

The actual throughput and latency measurements are done using a dedicated benchmarking .NET application whose source code can be found in the `src/supplementary/benchmark/ThroughputTests/` directory in the attachments of this thesis. The attachments also include a self-contained build of the application for both Windows and Linux operating systems in the `bin/{platform}/ThroughputTests/` directory. The self-contained application does not require .NET to be installed on the target machine. Note that on Linux, the `LD_LIBRARY_PATH` must be defined to the directory with the compiled binary for *MsQuic* to be loaded correctly.

The application implements a trivial echo server and clients which exchange messages of the specified size. The first parameter to the application selects one of the following modes:

`server`
: Starts only the server-side part of the application.

`client`
: Starts only the client-side part of the application. In this mode, the application spawns multiple clients and reports the measurement results on the standard output.

`inproc`
: Starts both the server and client in the same process. All network communication is done over the loopback network interface.

The benchmarking application accepts multiple parameters. A list of the most important follows. The full list of parameters can be found in the `Program.cs` file, or by running the program with the `--help` parameter.

`-e, --endpoint`
: The endpoint on which to listen (server) or to which to connect (client). Not applicable for `inproc` mode.

`-t, --tcp`
: Use TCP instead of QUIC.

`-c, --connections`
: The number of connetions to create.

`-s, --streams`
: The number of streams to create in each connection. Only for QUIC.

`-m, --message-size`
: The size of sent messages in bytes.

`-w, --warmup-time`
: Time before starting to take measurements. This can be used to give the application time to JIT all methods used on the hot path.

`-d, --test-duration`
: Time after which the measurement should stop and the application exit.

`-n, --no-wait`
: Whether clients wait for a reply before sending another message.

By default, the implementation uses managed QUIC implementation. Switching to *MsQuic*-based implementation is achieved by defining the `DOTNETQUIC_PROVIDER` environment variable to `msquic`.

The client mode of the application switches between two behaviors. By default, clients wait for a reply from the server and measure the delay until a reply is received. When using the `-n` switch, clients send as many messages as possible without waiting for the server and only count the number of replies. This lets us measure the total throughput of the system.

6.2 Measurement Results

In the following subsections, we present the results of the individual performance experiments. The throughput and latencies were measured as follows:

- *Throughput*: Throughput of the entire server. This is the total amount of application data echoed back to clients across all connections. The throughput measurements were taken with the use of `-n` flag in the benchmark application.
- *Latency*: The delay between client writing the message to the `Stream` and reading back the full reply. In our tests, we will measure the 99th percentile of latency.

We will perform three kinds of performance comparisons:

- *Multiple Parallel Streams Performance*: These tests compare how the managed and *MsQuic*-based QUIC implementations scale with increasing server load. These tests will utilize the stream multiplexing of QUIC to send messages using multiple parallel QUIC streams.
- *Single Stream Performance*: These tests use only a single stream in a connection. These tests should allow us to compare the performance between QUIC implementations and TCP+TLS-based `SslStream`.
- *Performance in Simulated Cellular Network*: In these tests, we will try to approximate the characteristics of a real-world cellular network by increasing lag and packet loss in the Linux LAN environment.

In all experiments in this section, client and server parts of the benchmarking application were run in separate processes. Each test case had a 5 s warm-up time and collected data for another 15 s. These intervals were long enough to produce stable measurements across multiple test runs.

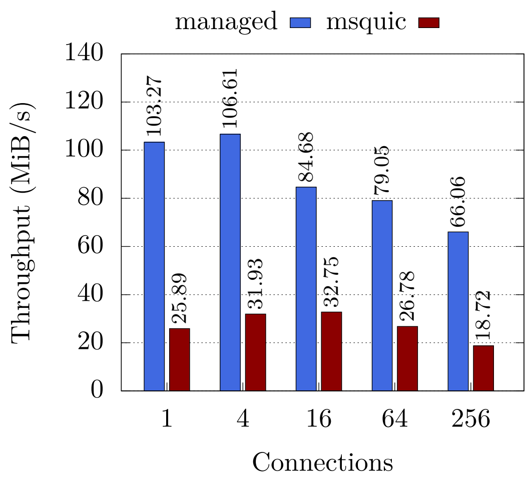
In the benchmarking application, clients and server exchange messages of the size specified by the `-m` parameter. In our tests, we will use mainly two message sizes: 256 B and 4096 B. The 256 B message are small enough to fit into a single QUIC packet, while the larger 4096 B messages are guaranteed to be split across multiple QUIC packets. We expect that increasing the message size should increase the latency because more packets need to be sent for both the message and the reply. Also, increasing the message should increase throughput because there are fewer calls to the `Write` and `WriteAsync` methods for the same amount of data.

Running the benchmarking application for tests in this section has been automated using a PowerShell [51] script. The script can be found at `src/supplementary/benchmark/ThroughputTests/run.ps1` in this thesis' attachments. Comments at the top of the file explain the usage of the script.

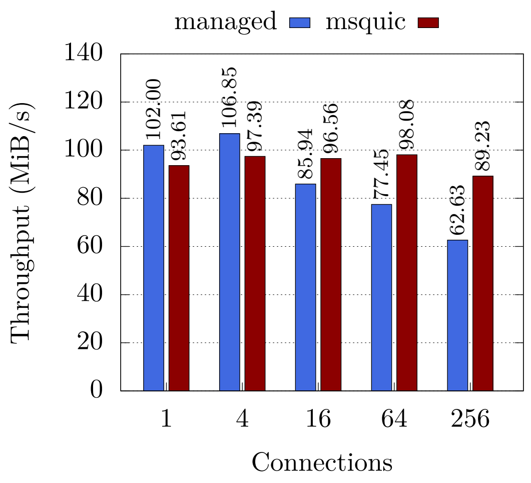
6.2.1 Multiple Parallel Streams Performance

The first set of tests compared the managed and *MsQuic*-based QUIC implementations. These tests were run with increasingly larger messages and with a greater number of parallel connections and streams to see how the two implementations scale.

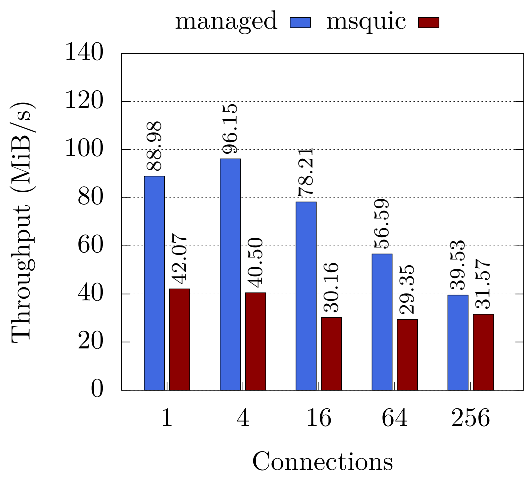
Figure 6.1 shows the measured throughput. Figure 6.1a shows the baseline performance when 256 B messages are sent using a single stream per connection. The other figures show measurements after increasing message size to 4096 B (Figure 6.1b), increasing number of streams to 32 (Figure 6.1c), or both (Figure 6.1d). An immediate observation can be made that the managed implementation produced very similar pattern in all four test runs, with the highest throughput at 4 parallel connections and then decreasing. On the other hand, the *MsQuic*-based implementation maintains almost the same throughput regardless of the number of connections, but increases throughput both when increasing the message size and number of streams in a connection.



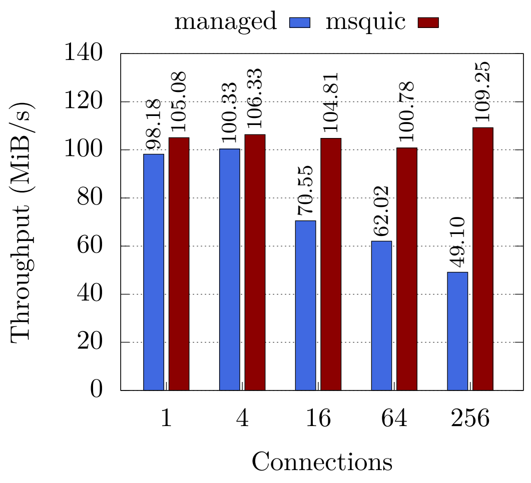
(a) 1 stream, 256 B messages



(b) 1 stream, 4096 B messages



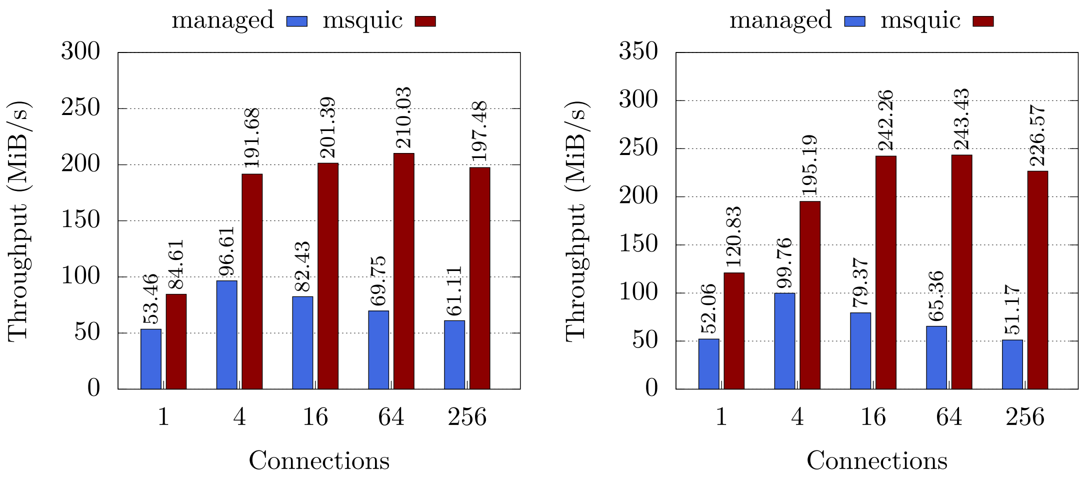
(c) 32 stream, 256 B messages



(d) 32 stream, 4096 B messages

Figure 6.1: Multiple stream QUIC throughput measurements (Linux LAN)

In case with 4096 B messages, the *MsQuic*-based implementation consistently saturates the 1 Gbit network connection between the two blade servers. To give more perspective on the relative performance between the two implementations, Figure 6.2 shows results of the two test runs on the Windows workstation over the loopback interface. Depending on the number of connections, the throughput of the *MsQuic*-based implementation was up to four times greater than our implementation. Results of runs with 256 B messages were left out for brevity because they did not differ from the ones in the Linux LAN environment.



(a) 1 stream, 4096 B messages **(b)** 32 stream, 4096 B messages

Figure 6.2: Multiple stream QUIC throughput measurements (Win loopback)

The peak of the throughput of our managed QUIC implementation at four connections can be explained by the fact that we are spawning a long-running background task for each connection. Four connections lead to a number of parallel `Task`s that were enough to completely utilize the entire CPU.

As for the decline of the throughput of managed QUIC implementation when 16 or more connections were used, profiling showed that a substantial amount of time was spent in the garbage collection. At 256 connections, up to 50% of the total CPU time was spent in GC, divided into relatively large pauses of 20 ms or more. The pauses introduced by GC led to QUIC packets being considered lost, which led to the collapse of the congestion window in the QUIC connection, further reducing the rate at which data were sent. This points to the fact that even though we carefully avoided needless allocations in our implementation, there are still enough sources of allocation left to degrade a server's performance under heavy load.

An example of a large allocation source in the managed QUIC implementation is pooling the buffers for sending or receiving QUIC packets. Our implementation uses `ArrayPool<byte>.Shared` to reuse allocated buffers. However, it does not work well with a large number of connections because `ArrayPool<byte>.Shared` pools only a few `byte[]` instances and lets GC collect the rest[3]. With a large num-

[3]In .NET 5, the implementation of `ArrayPool<byte>.Shared` maintains a separate pool for each CPU core. Each per-CPU core pool organizes pooled arrays into 17 buckets with array sizes ranging from 16 B to 1 MiB. Each bucket retains up to 8 array instances of similar size.

ber of connections, the buffers are rented and returned in large bursts, leading to a lot of large arrays being discarded upon return to the `ArrayPool<byte>.Shared` and subsequently allocated anew.

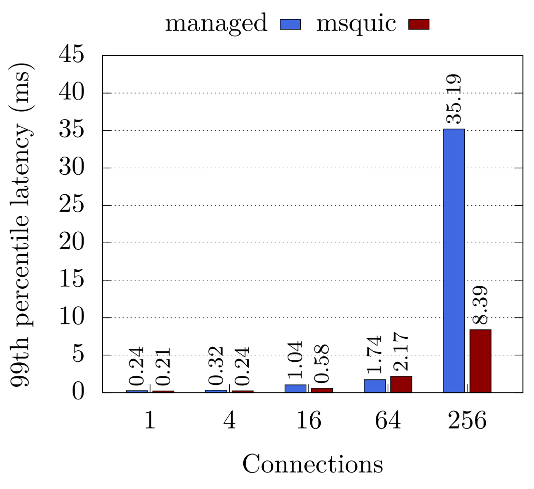
Another source of allocations are `Socket.SendTo` and `Socket.ReceiveFrom` methods which allocate of a new `EndPoint` instance for each call. In order to remove this particular source of allocations, an allocation-free API for sending or receiving UDP datagrams needs to be designed and implemented on the `Socket` class. Such API has already been proposed in the past [52] but has not been prioritized to be implemented.

MsQuic-based implementation shows similar throughput regardless of the number of parallel connections. Instead, it dramatically increases with message size and only slightly with the number of streams in the connections. When we tried sending messages larger than 4096 B, the total throughput did not increase significantly anymore. Upon closer inspection of the *MsQuic* network traffic using Wireshark [35], it turns out that in tests with 256 B messages, many of the sent packets were very small, possibly containing only `ACK` frames. Thus, the total available network bandwidth was mostly unutilized. When 4096 B messages are used, more space in QUIC packets is utilized, which increases the throughput. Our investigation, however, did not uncover why the total throughput of *MsQuic* does not increase with the number of connections in the tests with 256 B message.

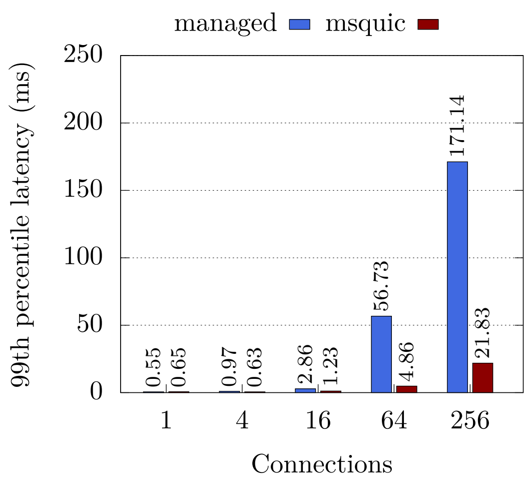
Figure 6.3 shows the latencies measured in the Linux LAN environment for the same four test cases as above. In all tests, the *MsQuic*-based implementation exhibits a lower 99th percentile of latency, especially in test cases with a high number of connections. The same measurement on Windows loopback did not yield significantly different results.

When measuring latency, the benchmarking application waits for the server to respond before sending another message. However, with many parallel connections and streams, the total network traffic becomes similar to that produced when measuring throughput. This was the case, especially when we see significant differences between the latencies of the two implementations. Therefore, the change in latency can be attributed to the frequent and long pauses introduced by the GC that we have described above. On the other hand, in the test runs where the managed implementation is close to that of *MsQuic*, the GC activity was under 5% and did not cause long frequent pauses.

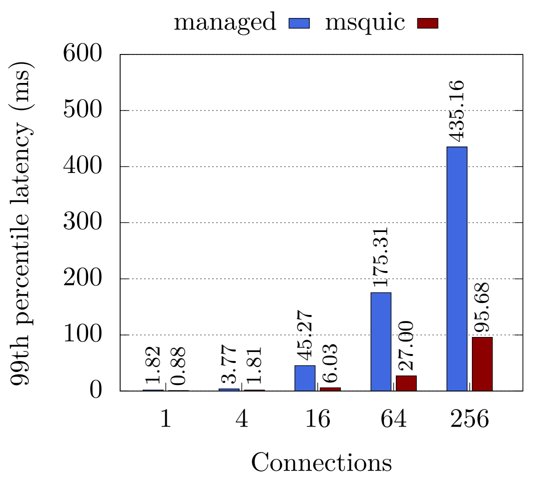
By increasing the message size, we increased the probability that the application message will be affected by the packet loss introduced by the GC pauses. This happens because the message is spread over multiple QUIC packets, and the loss of any of those packets will delay the entire message until the missing part is retransmitted. This explains why increasing the message size drastically affects the latency with a large number of parallel connections.



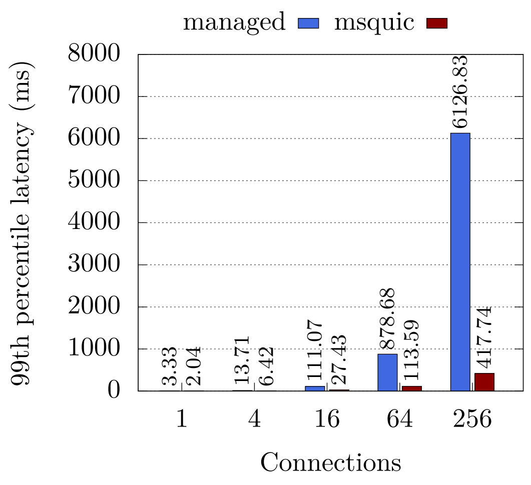
(a) 1 stream, 256 B messages



(b) 1 stream, 4096 B messages



(c) 32 stream, 256 B messages

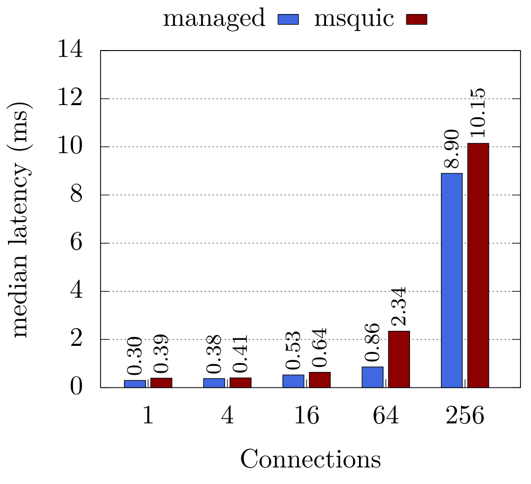


(d) 32 stream, 4096 B messages

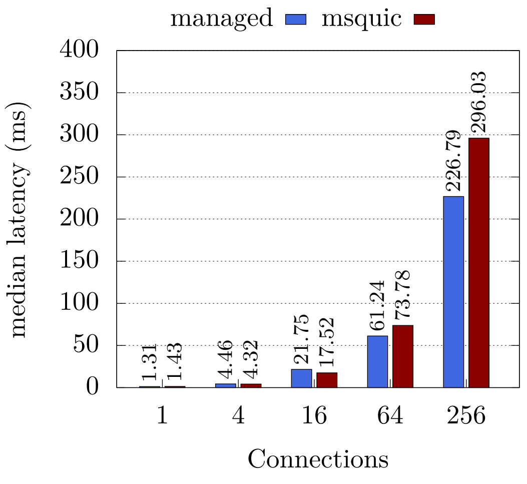
Figure 6.3: Multiple stream QUIC latency measurements (Linux LAN)

Increasing the number of streams also increases network traffic, making our implementation susceptible to the increased GC pauses. Also, after a certain threshold, increasing the number of streams does not increase the traffic because the maximum throughput of the implementation has been reached. Instead, it leads to an increase in latency because it still increases the number of outstanding parallel requests.

Lastly, we would like to remind the reader that we measured the 99th percentile of the latency, which considers only the slowest 1% of requests. To give more perspective on the latency distribution on the two implementations, Figure 6.4 shows the median measurements of the latency. For brevity, we include only the measurements for the cases with 4096 B messages. The median measurements of latencies of our implementation are similar or lower than that of the *`MsQuic`*-based one.



(a) 1 stream, 4096 B messages



(b) 32 stream, 4096 B messages

Figure 6.4: Multiple stream QUIC latency median measurements (Linux LAN)

We believe that the difference in the latency distributions between the two implementations is due to their different architectures. *MsQuic* spawns several worker threads (depending on the number of CPU cores) which process events for individual connections from an event queue. The event queue ensures that the connections are serviced evenly and at regular intervals. When the load on the server increases, the extra latency is spread evenly across all active connections. This is further supported by the fact that the median values for the latency of *MsQuic* are only slightly lower than its 99th percentile from Figure 6.3.

Our implementation, on the other hand, relies solely on the .NET `Task` scheduler, which schedules all background tasks independently and without any priority information. This allows for greater variance in the latency of our QUIC implementation as replies can be delayed due to unfortunate scheduling. In the future, our implementation's architecture should be improved to reduce the scheduling variance and, therefore, the 99th percentile of the latency.

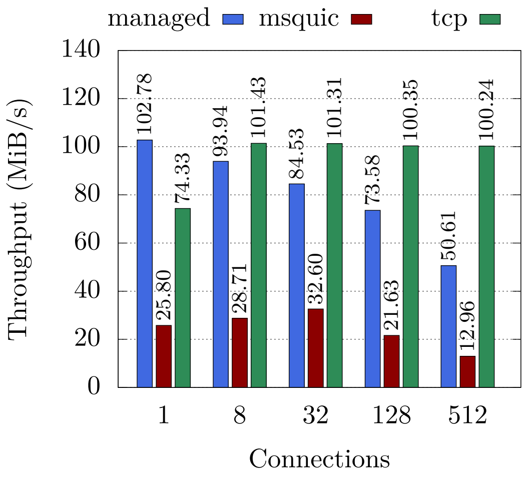
In summary, our QUIC implementation outperforms the *MsQuic*-based one in scenarios where small messages are exchanged between client and server using a small number of streams. However, our implementation does not scale as well as the *MsQuic*-based one. A large number of parallel connections and streams increases pressure on the GC, which introduces frequent and long pauses in the application. Lastly, our implementation's latency distribution is much less uniform than that of the *MsQuic*-based one because of the way background work is scheduled on the .NET thread-pool.

6.2.2 Single Stream Performance

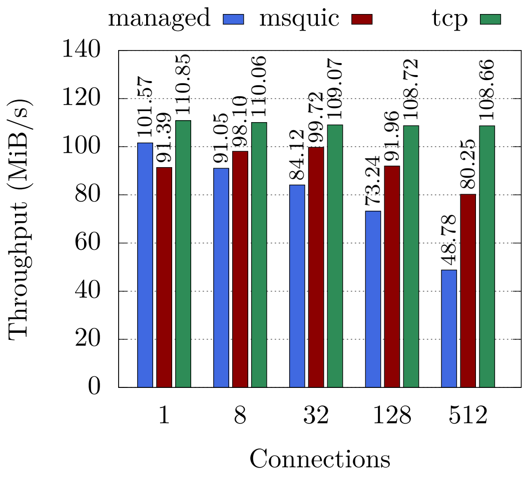
In the second test, we compared the single-stream performance of managed QUIC implementations to the performance of the combination of TCP and TLS available via the `TcpClient` and `SslStream` .NET classes. We will also include the measurements for *MsQuic* for completeness.

Figure 6.5 shows the results of the throughput measurements taken between the two blade servers connected by 1 Gbit Ethernet. In both 256 B messages

(Figure 6.5a) and 4096 B messages (Figure 6.5b) scenario, TCP implementation outperforms both QUIC implementations. Also, out of the three implementations, our QUIC implementation is the only one which is not bottlenecked by the 1 Gbit LAN.



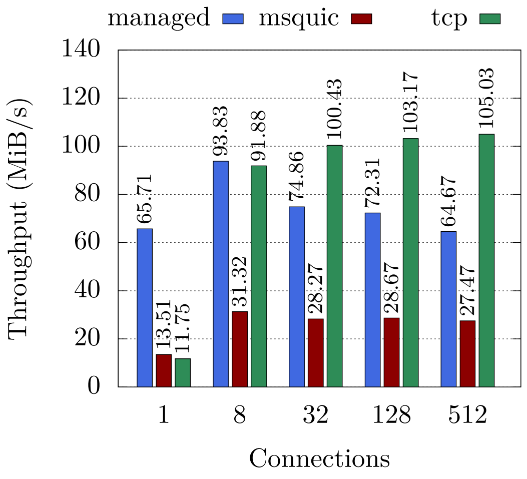
(a) 256 B messages



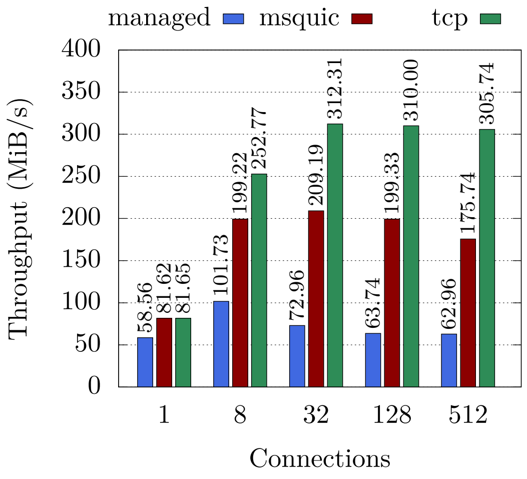
(b) 4096 B messages

Figure 6.5: Single stream throughput measurements (Linux LAN)

Because the throughput of some implementations in the previous figure was limited by the 1 Gbit network, Figure 6.6 shows the results measured over the loopback interface on the Windows workstation. We can see clearly that TCP still provides greater throughput than the *MsQuic*-based implementation. We would like to remind the reader that these measurements were taken with the size of the IP datagram limited to 1500 B (see section 6.1.1). Otherwise TCP would send the maximum 64 KiB packets and would reach throughput up to 900 MiB/s.



(a) 256 B messages

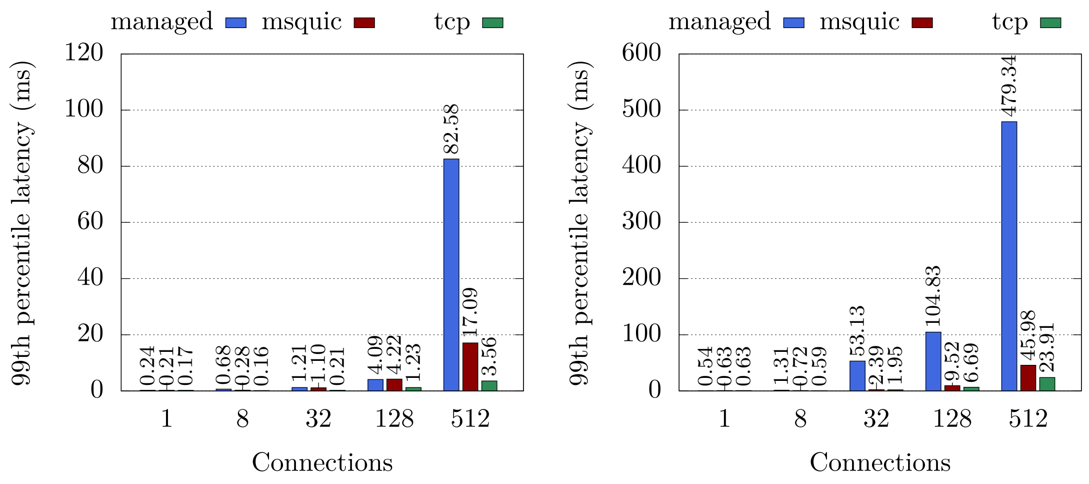


(b) 4096 B messages

Figure 6.6: Single stream throughput measurements (Win loopback)

By comparing the numbers in Figure 6.6b to that from Figure 6.2b, we see that the performance of TCP is also greater than that of *`MsQuic`*-based implementation when using 32 parallel streams per connection streams.

Figure 6.7 shows the latency measurement results in the Linux LAN environment. These results show that TCP exhibits substantially lower latencies than either QUIC implementation. Very similar results were also obtained on the Windows workstation.



(a) 256 B messages **(b)** 4096 B messages

Figure 6.7: Single stream latency measurements (Linux LAN)

It is worth noticing that in Figure 6.7a, the latency of managed QUIC implementation is very low until it reaches a certain threshold and then increases rapidly as a consequence of increased GC pauses.

The results presented in this section suggest that neither QUIC implementation can compete with the `SslStream` in neither local LAN networks nor over the loopback interface (even with the MTU reduction we explained in section 6.1.1). It is possible, however unlikely, that the performance over real 10 Gbit/s network will yield different results. A possible explanation for these results is that the TCP networking stack is far better optimized than UDP. Also, in the Linux LAN environment, some hardware optimizations may have been in place for TCP, such as *TCP offload engine* [53] which offloads the TCP/IP stack implementation onto the controller on the network interface card.

6.2.3 Performance in Simulated Cellular Network

In the last test, we compared the resilience of QUIC and TCP+TLS implementations in a simulated network with delay and packet loss. When choosing the values of delay and packet loss parameters, we tried to approximate a 4G cellular broadband network's behavior. We performed a trivial connection speed test of local 4G network using a smartphone and the `speedtest.net` testing web application. This test reported latency of 25 ms, which we then chose as the latency in the simulated network in our tests.

As for the packet loss in 4G network, in 2012, Chen et al. have measured characteristics of 4G networks of major US cellular network carriers and measured packet loss percentages from 0.004% to 0.1% [54]. Even though the cellular network technology may have evolved since then, we could not find more recent measurements. Therefore, we will use those two packet loss values in our test parameters.

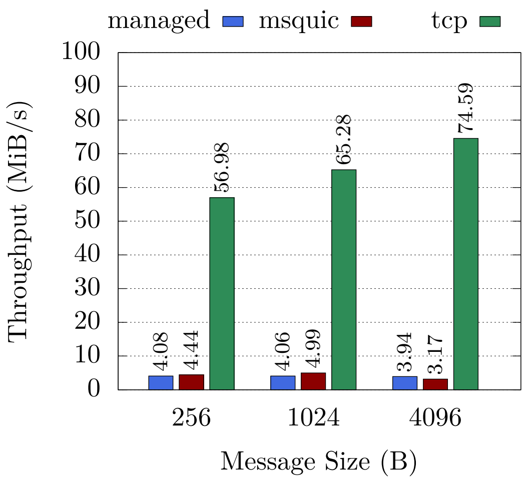
It should be noted that our approximation of the cellular network is not perfect. The latency in a real-world network is not constant but has a random distribution, and the 25 ms we measured earlier was only the mean value. Similarly, packet loss does not affect each packet independently. Instead, packets are often lost in bursts. There are multiple models for modeling more realistic packet loss, such as the Gilbert-Elliot model [55]. However, we were unable to find enough data to compute appropriate parameters for such models.

To simulate the 4G network, we used the *Traffic Control* capabilities of the Linux kernel on the two blade servers in the Linux LAN evaluation environment. The necessary configuration can be set using the `tc` utility. For example, the following command issued on both machines would simulate 25 ms lag and 0.004% packet loss:

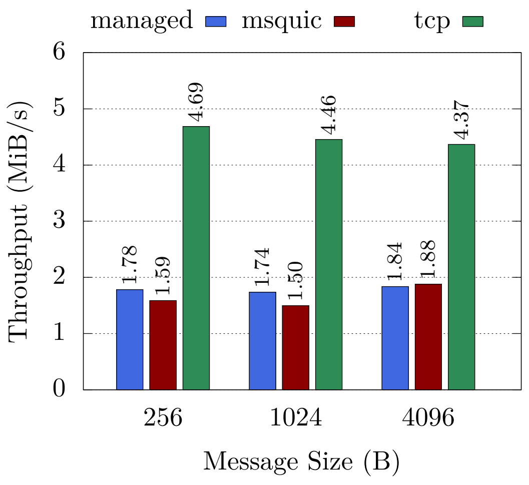
```
tc qdisc change dev eth0 root netem delay 12.5ms loss 0.004%
```

Note that the set delay parameter is half of the desired latency because the traffic shaping done by `tc` applies only to outbound traffic. Each endpoint, therefore, adds half of the total latency.

Figure 6.8 shows the measured throughput for 0.004% packet loss (Figure 6.8a) and 0.1% packet loss (Figure 6.8b). As expected, The increase in packet loss leads to a decrease in throughput. In the presence of only low packet loss, the TCP implementation exhibits an order of magnitude greater throughput. However, the throughput of the TCP protocol decreased at a greater rate than that of the two QUIC implementation. The throughput of the two QUIC implementation is comparable in all tests.



(a) 25 ms latency, 0.004% loss

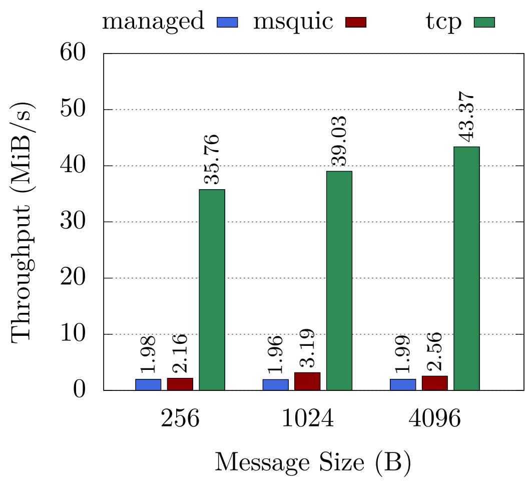


(b) 25 ms latency, 0.1% loss

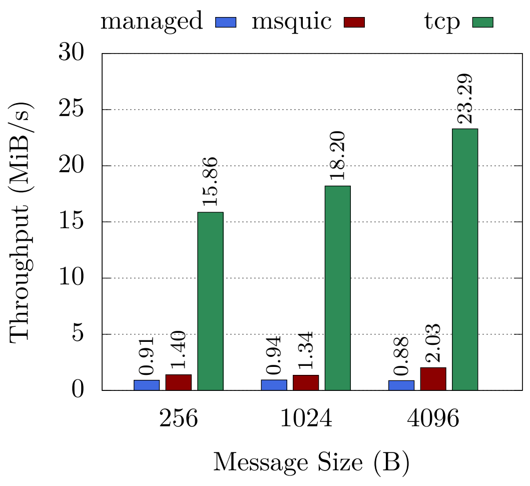
Figure 6.8: Single stream throughput measurements in simulated network (Linux LAN)

Upon closer inspection, it turns out that the managed QUIC implementation is bottlenecked by the stream buffering implementation. In section 3.6.2, we explained the design behind the sending part of the stream, namely the `Send Stream` class. The implementation uses up to eight 16 KiB buffers to store the application data. Once these buffers are filled, subsequent calls to the `Write` or `WriteAsync` method will block until some data are acknowledged. The application data must be buffered until the other endpoint acknowledges their reception. This takes at least one roundtrip, i.e., 25 ms in our test. Our implementation can, therefore, send at most $8 \cdot 16 = 128\text{KiB}$ on a single stream during a round trip period. This sending rate implies a theoretical maximum throughput as $128\text{KiB} \div 0.025 = 5\text{MiB/s}$.

This hypothesis can be tested by doubling the delay on the simulated network. Figure 6.9 shows measurements with 50 ms latencies (Figure 6.9a) and 100 ms latencies (Figure 6.9b). In these variations of the test we see that the throughput of the managed implementation is highly correlated with the network latency as it halves everytime the latency is doubled.



(a) 50 ms latency, 0.004% loss

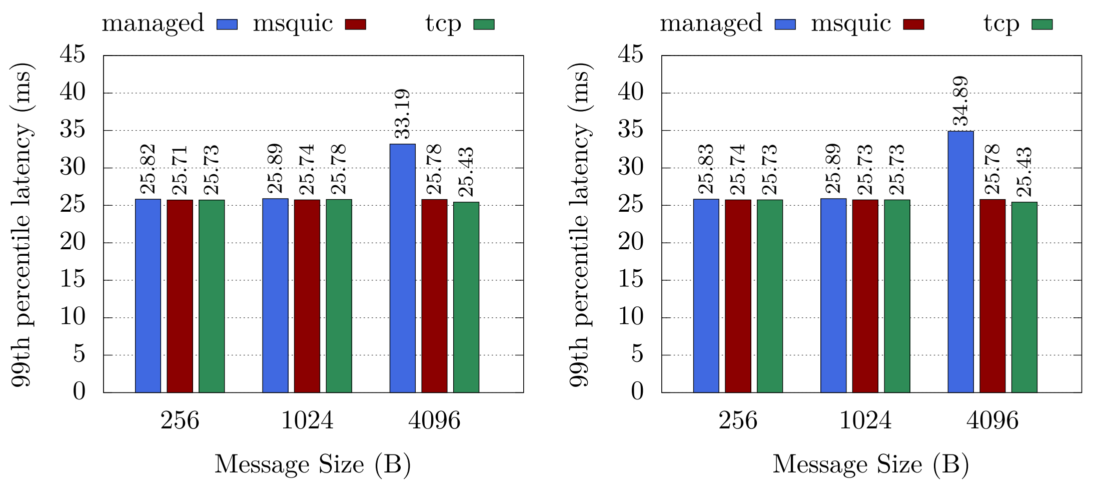


(b) 100 ms latency, 0.004% loss

Figure 6.9: Single stream throughput measurements in simulated network with long delays (Linux LAN)

This performance limitation could be fixed simply by increasing the number of buffers used to buffer application data. However, a large number of such buffers would lead to inefficient memory utilization in low-latency environments like a LAN network. Instead, the amount of data buffered should be changed dynamically based on information such as the congestion window's size or the flow control limit advertised by the other endpoint. We leave the implementation of such dynamic buffering for future work.

Figure 6.10 shows the latency measurements results. Each implementation exhibits the same latency for both 0.004% (Figure 6.10a) and 0.1% (Figure 6.10b) packet loss. This means that even 0.1% packet loss rate is still low enough to not affect the 99th percentile of latency.



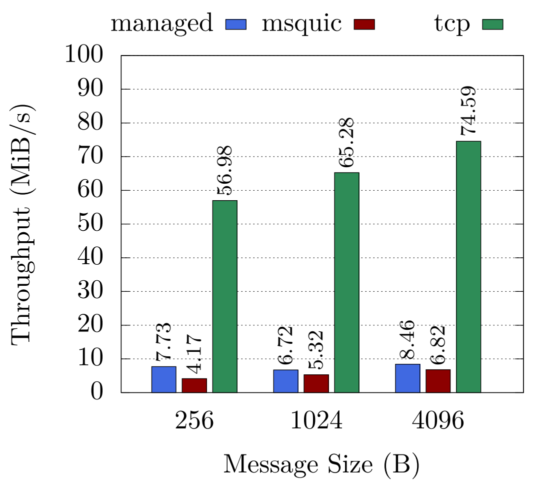
(a) 25 ms latency, 0.004% loss **(b)** 25 ms latency, 0.1% loss

Figure 6.10: Single stream latency measurements in simulated network (Linux LAN)

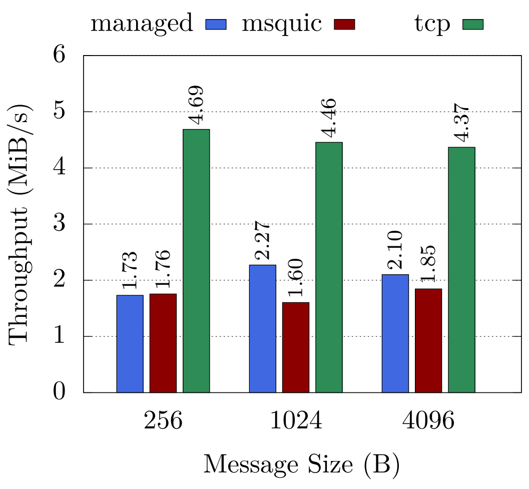
All implementations show almost the same latency in almost all tests. The single exception is our implementation, which exhibits greater latencies for 4096 B messages. This is because this message size does not fit a single packet and our packet pacing implementation (see section 4.4.3) spaces out individual packets one by one and thus increases the delay between sending the first and last QUIC packet carrying the 4096 B message. The *MsQuic*-based implementation, on the other hand, sends small batches of packets for each tick of the pacer and the small batches are enough to transmit the entire 4096 B message.

By using only a single stream in each QUIC connection in these tests, it could be argued that we did not utilize one of the big advantages of QUIC over TCP — the reduced head-of-line blocking. Figure 6.11 shows the measurements of throughput like we did previously (for Figure 6.8), but with QUIC connections transmitting data using 32 parallel streams. Contrary to the expectation, the throughput of the QUIC implementations increases only slightly.

Because we were sending data using multiple streams, the managed QUIC implementation was no longer blocked by the limited buffering discussed above. Instead, investigation shows that it was limited by the size of the congestion window. After more investigation, it turned out that the throughput of both QUIC implementation was very high at the beginning, possibly even higher than that of TCP. However, during the 5 s warmup period, the packet loss led to a significant reduction of the congestion window, and the throughput was reduced to the values we can see on the plot.



(a) 32 streams, 25 ms latency, 0.004% loss



(b) 32 streams, 25 ms latency, 0.1% loss

Figure 6.11: Multiple stream throughput measurements in simulated network (Linux LAN)

We believe that this behavior was caused by the too uniform distribution of the packet loss we used in our testing environment. In an environment with uniform packet loss, almost all lost packets induce a separate congestion event and reducing the congestion window. When packets are lost in bursts — as is more common in real-world networks — the loss of multiple consecutive packets is handled as a single congestion event, leading to less frequent reductions of the congestion window.

In this case, the protocol's behavior depends on the implementation of the congestion control algorithm. The managed QUIC implementation uses an adaptation of the NewReno algorithm described by the QUIC specification [20, Section 7]. The *`MsQuic`*-based QUIC implementation and TCP used the CUBIC algorithm [56]. Because of the substantial difference in performance, we consider it worthwhile to experiment with different congestion control algorithms for our implementation. Studying the behavior and implementation of TCP congestion control in the Linux kernel should also provide useful insight for improving our implementation's behavior. However, due to the complexity, we leave such experiments and investigations for future work.

6.3 Interop between QUIC Implementations

The benchmarking application can also test the managed QUIC implementation's interoperability with the *`MsQuic`*-based implementation. This can be done by starting one application in `server` mode using one implementation provider and another instance of the application in `client` mode using the other provider.

We have tested both combination with following results:

- *Managed client and `MsQuic` server*: With a small number of connections, the two implementations interoperated without any noticeable problems. However, starting the benchmarking application with a very large number of parallel connection sometimes made *`MsQuic`* server respond with a

Retry packet for some connections. This is because `MsQuic` engages client address validation (see section 2.8.3) if there are too many simultaneous connections attempts. Our implementation does not implement support for client address validation via Retry packets, and, therefore, the connection was terminated immediately.

- *`MsQuic` client and Managed server*: The version of the `MsQuic` library we used in our tests started all client connections using the draft 29 version of the protocol, while our implementation supports only the draft 27 version. Because we did not implement version negotiation, the server did not attempt negotiating the draft 27 protocol version, and the `MsQuic` client connections eventually closed due to idle timeout. Even if we decided to ignore the protocol version in our implementation, these two versions use a different *initial salt* for deriving Initial packet protection keys (see section 2.8.2). Therefore, our server would drop all incoming packets because they could not be decrypted.

Because the only problems with interop between the two QUIC implementations were due to features we explicitly decided to omit from our prototype implementation during our analysis in section 3.1, we are satisfied with these results.

Our benchmarking application did not perform an exhaustive interop test. A better interop tests could be achieved using the open-source QUIC Interop Test Runner project [57] which tests interoperabilty of popular QUIC implementations in various scenarios. However, this level of testing is outside of the scope of this thesis and is subject to future work.

6.4 Summary

Compared to `MsQuic`-based QUIC implementation provider, the managed QUIC implementation developed as part of this thesis can provide higher throughput when using a small number of parallel connections while maintaining comparable latencies. However, in the current state, the managed QUIC implementation does not scale very well, and both throughput and observed latencies degrade when large number of connections are created. When under substantial load, the `MsQuic`-based implementation delivered throughput up to four times greater than our implementation.

However, in our tests, the TCP+TLS stack has significantly larger throughput and lower latencies than either QUIC implementation in our LAN evaluation environment. A possible explanation of this is the presence of hardware optimizations for the TCP stack on the network interface card.

We also compared the QUIC and TCP implementations in a simulated 4G network. In our tests, we again measured significantly higher throughput using TCP+TLS stack than either QUIC implementation, even if we used multiple streams in QUIC connections. As for latencies, all implementations exhibit almost identical latency, except for our implementation, which had larger latencies once the messages exceeded the maximum size of a QUIC packet.

From our measurements, we conclude that our prototype QUIC implementation will need further development and performance tuning in order to provide performance similar to that of `MsQuic`. We have identified some of the underlying problems in the current managed implementation and proposed future improvements. However, both QUIC implementations will require a significant amount of optimizations before reaching performance comparable to present TCP implementations.

Conclusion

To conclude our thesis, we will revisit the goals we set in the introduction chapter in section 1.4.

1. *Select a sufficient subset of QUIC specification needed to support the most basic data transfer and implement it inside .NET runtime codebase.*

   In section 3.1, we analyzed parts of the QUIC protocol and selected the necessary parts for our prototype implementation. All features selected in this section were implemented to the extent that the implementation can successfully and reliably transfer data over the network.

2. *Allow switching between the new managed implementation and the existing `MsQuic`-based one.*

   Our implementation integrates into the pre-existing implementation indirection layer, which allows explicitly selecting the QUIC implementation provider for newly created `QuicListener` and `QuicConnection` instances. Additionally, the default provider can be selected using the `DOTNETQUIC_PROVIDER` environment variable.

3. *Evaluate the managed QUIC implementation by using it to implement a simple client-server echo application.*

   In section 5.2, we provided directions on implementing a simple echo server and client. Additionally, we implemented a benchmarking application for use in performance measurements in chapter 6.

4. *(optional) Compare the performance of the new implementation with the previous `MsQuic`-based one and with TCP+TLS-based `SslStream`.*

   In section 6.2, we presented the results of our performance measurements. Our QUIC implementation outperforms the `MsQuic`-based one in a small number of scenarios. However, in most cases and especially under heavy load, the `MsQuic`-based QUIC implementation performs better both in terms of throughput and latency. However, neither of the QUIC implementations performed better in our tests than the combination of TCP+TLS.

At the time of writing this conclusion, our managed QUIC implementation has caught the attention of the .NET development team, and this implementation will be added to the list of experiments in the runtimelab repository [58] in the `feature/ManagedQuic` branch.

The next big release — .NET 6 — will ship with production-ready QUIC implementation. In early 2021, a decision will be made whether this QUIC support will be based on `MsQuic` or our QUIC implementation. However, even if the more mature `MsQuic` implementation is chosen for the .NET 6 release, our implementation will be considered as a managed replacement for subsequent .NET releases.

Future Work

The prototype QUIC implementation developed in this thesis will require more development and substantial performance tuning before becoming production-ready. Some parts of the QUIC specification were left unimplemented, and other parts were simplified to fit into the scope of this master thesis. However, the core part of the implementation should provide a solid foundation upon which a fully conformant QUIC implementation could be built. The following list outlines the next development steps for the implementation.

- *Update the implementation to match the latest QUIC specification.* QUIC specification drafts evolved both during implementation and writing of the text of this thesis. At the time of writing this conclusion, the 33rd version of the QUIC specification draft is awaiting a last-call before it becomes a valid RFC document. The implementation presented in this document is based on draft version 27. Updating the implementation should be, therefore, the first future goal.

- *Implement missing parts of the protocol.* This thesis implements only a subset of the QUIC specification. Many features like connection migration, stateless reset, and path validation have been omitted from the prototype but should be implemented to make the implementation fully conform to the QUIC specification.

- *Performance improvements for scalability.* The performance measurements done in section 6.2.1 show that our implementation does not scale very well in the face of large amounts of parallel connections. Possible improvements to the backend processing architecture include allowing parallel sending and receiving on a single connection and using a single thread to process multiple connections like done in ***MsQuic***.

- *Experiment with congestion control algorithms.* This thesis implemented only the NewReno [20, Section 7] congestion control algorithm described in the QUIC protocol specification. However, other algorithms such as CUBIC or HyStart++ [32] have shown better performance in some scenarios.

- *More realistic performance measurements.* This thesis performed measurements using 1 Gbit/s Ethernet connection in LAN network. Some implementations easily saturated such a network, and we had to extrapolate from results measured over the software-based loopback network interface. It would be better to confirm the measurements over 10 Gbit/s connection.

- *Interoperability tests with other QUIC implementations.* This thesis has done only a straightforward interoperability test with ***MsQuic***. There is an open-source QUIC Interop Test Runner [57] which repeatedly tests compatibility between the latest versions of popular QUIC implementations.

Bibliography

[1] Mike Belshe, Roberto Peon, and Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. RFC Editor, May 2015. 96 pp. DOI: 10.17487/RFC7540. URL: https://rfc-editor.org/rfc/rfc7540.txt.

[2] Roy T. Fielding and Julian Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. RFC Editor, June 2014. 89 pp. DOI: 10.17487/RFC7230. URL: https://rfc-editor.org/rfc/rfc7230.txt.

[3] H. de Saxcé, I. Oprescu, and Y. Chen. "Is HTTP/2 really faster than HTTP/1.1?" In: *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 2015, pp. 293–299.

[4] Wikipedia contributors. *Head-of-line blocking — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Head-of-line_blocking (visited on 08/23/2020).

[5] Eric Rescorla. *HTTP Over TLS*. RFC 2818. RFC Editor, May 2000. 7 pp. DOI: 10.17487/RFC2818. URL: https://rfc-editor.org/rfc/rfc2818.txt.

[6] *HTTPS encryption on the web - Google Transparency Report*. URL: https://transparencyreport.google.com/https/overview (visited on 08/11/2020).

[7] Mike Bishop. *Hypertext Transfer Protocol Version 3 (HTTP/3)*. Internet-Draft draft-ietf-quic-http-27. Work in Progress. Internet Engineering Task Force, Feb. 2020. 61 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-27.

[8] Alessandro Ghedini. *Even faster connection establishment with QUIC 0-RTT resumption*. Ed. by Cloudflare. Nov. 20, 2019. URL: https://blog.cloudflare.com/even-faster-connection-establishment-with-quic-0-rtt-resumption/.

[9] Quentin De Coninck and Olivier Bonaventure. *Multipath Extensions for QUIC (MP-QUIC)*. Internet-Draft draft-deconinck-quic-multipath-04. Work in Progress. Internet Engineering Task Force, Mar. 2020. 36 pp. URL: https://datatracker.ietf.org/doc/html/draft-deconinck-quic-multipath-04.

[10] Alyssa Wilk, Ryan Hamilton, and Ian Swett. *A QUIC update on Google's experimental transport*. Apr. 17, 2015. URL: https://blog.chromium.org/2015/04/a-quic-update-on-googles-experimental.html.

[11] Sreeni Tellakula. *Comparing HTTP/3 vs. HTTP/2 Performance*. Apr. 14, 2020. URL: https://blog.cloudflare.com/http-3-vs-http-2/.

[12] Sarah Cook et al. "QUIC: Better for what and for whom?" In: *2017 IEEE International Conference on Communications (ICC)*. May 2017, pp. 1–6. DOI: 10.1109/ICC.2017.7997281.

[13] *HttpClient Class*. Microsoft. URL: https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient (visited on 08/19/2020).

[14] Microsoft. *MsQuic - Cross-platform, C implementation of the IETF QUIC protocol.* URL: https://github.com/microsoft/msquic (visited on 08/14/2020).

[15] Daniel Stenberg. *Curl - A command line tool and library for transferring data with URL syntax.* 2020. URL: https://github.com/curl/curl (visited on 08/14/2020).

[16] *SocketsHttpHandler Class.* Microsoft. URL: https://docs.microsoft.com/en-us/dotnet/api/system.net.http.socketshttphandler (visited on 08/19/2020).

[17] .NET Platform. *.NET Runtime.* URL: https://github.com/dotnet/runtime (visited on 08/14/2020).

[18] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport.* Internet-Draft draft-ietf-quic-transport-27. Work in Progress. Internet Engineering Task Force, Feb. 2020. 174 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-27.

[19] Martin Thomson and Sean Turner. *Using TLS to Secure QUIC.* Internet-Draft draft-ietf-quic-tls-27. Work in Progress. Internet Engineering Task Force, Feb. 2020. 52 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-tls-27.

[20] Jana Iyengar and Ian Swett. *QUIC Loss Detection and Congestion Control.* Internet-Draft draft-ietf-quic-recovery-27. Work in Progress. Internet Engineering Task Force, Feb. 2020. 41 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-recovery-27.

[21] Andrei Gurtov et al. *The NewReno Modification to TCP's Fast Recovery Algorithm.* RFC 6582. RFC Editor, Apr. 2012. 16 pp. DOI: 10.17487/RFC6582. URL: https://rfc-editor.org/rfc/rfc6582.txt.

[22] Stephan Friedl et al. *Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension.* Tech. rep. 7301. RFC Editor, July 2014. 9 pp. DOI: 10.17487/RFC7301. URL: https://rfc-editor.org/rfc/rfc7301.txt.

[23] Donald E. Eastlake 3rd. *Transport Layer Security (TLS) Extensions: Extension Definitions.* RFC 6066. RFC Editor, Jan. 2011. 25 pp. DOI: 10.17487/RFC6066. URL: https://rfc-editor.org/rfc/rfc6066.txt.

[24] David McGrew. *An Interface and Algorithms for Authenticated Encryption.* RFC 5116. RFC Editor, Jan. 2008. 22 pp. DOI: 10.17487/RFC5116. URL: https://rfc-editor.org/rfc/rfc5116.txt.

[25] Dr. Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF).* RFC 5869. RFC Editor, May 2010. 14 pp. DOI: 10.17487/RFC5869. URL: https://rfc-editor.org/rfc/rfc5869.txt.

[26] Justin Kotalik. *[QUIC] Design of Shutdown for QuicStreams in System.Net.Quic.* Dec. 11, 2019. URL: https://github.com/dotnet/runtime/issues/756.

[27] *Schannel - Win32 apps*. Microsoft. URL: https://docs.microsoft.com/en-us/windows/win32/com/schannel (visited on 09/12/2020).

[28] OpenSSL Software Foundation. *OpenSSL - Cryptography and TLS/SSL toolkit*. URL: https://www.openssl.org/ (visited on 09/13/2020).

[29] OpenSSL Management Committee. *QUIC and OpenSSL*. Feb. 17, 2020. URL: https://www.openssl.org/blog/blog/2020/02/17/QUIC-and-OpenSSL/.

[30] Akamai Technologies, Inc. *TLS/SSL and crypto library*. https://github.com/akamai/openssl. Forked from https://github.com/openssl/openssl. Sept. 13, 2020.

[31] Cloudflare. *Quiche - Savoury implementation of the QUIC transport protocol and HTTP/3*. 2020. URL: https://github.com/cloudflare/quiche (visited on 11/02/2020).

[32] Junho Choi. *CUBIC and HyStart++ Support in quiche*. Aug. 5, 2020. URL: https://blog.cloudflare.com/cubic-and-hystart-support-in-quiche/.

[33] Wikipedia contributors. *Strategy pattern — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Strategy_pattern (visited on 11/13/2020).

[34] .NET Foundation. *xUnit.net*. URL: https://xunit.net/ (visited on 11/25/2020).

[35] Wireshark Foundation. *Wireshark*. 2020. URL: https://www.wireshark.org/.

[36] Robin Marx. *qvis: tools and visualizations for QUIC and HTTP/3*. URL: https://qvis.edm.uhasselt.be/ (visited on 10/18/2020).

[37] Robin Marx. *QUIClog - Tools and definitions for QUIC implementation debugging and logging*. https://github.com/quiclog. Oct. 18, 2020.

[38] Wikipedia contributors. *Abstract factory pattern — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Abstract_factory_pattern (visited on 11/17/2020).

[39] Elinor Fung. *Improvements in native code interop in .NET 5.0*. Sept. 1, 2020. URL: https://devblogs.microsoft.com/dotnet/improvements-in-native-code-interop-in-net-5-0/.

[40] Radek Zikmund. *.NET Runtime - Managed QUIC Implementation*. URL: https://github.com/rzikm/runtimelab/tree/master-managed-quic.

[41] *Reference assemblies*. Microsoft. URL: https://docs.microsoft.com/en-us/dotnet/standard/assembly/reference-assemblies (visited on 12/21/2020).

[42] .NET Platform. *.NET SDK installer*. URL: https://github.com/dotnet/installer (visited on 11/25/2020).

[43] *Dynamic-Link Library Search Order*. Microsoft. URL: https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-search-order#search-order-for-desktop-applications (visited on 12/15/2020).

[44] *ld.so(8) — Linux manual page*. The Linux man-pages project. URL: https://man7.org/linux/man-pages/man8/ld.so.8.html (visited on 12/15/2020).

[45] *Application publishing - .NET*. Microsoft. URL: https://docs.microsoft.com/en-us/dotnet/core/deploying (visited on 12/21/2020).

[46] *.NET Core Runtime Identifier Catalog*. Microsoft. URL: https://docs.microsoft.com/en-us/dotnet/core/rid-catalog (visited on 08/19/2020).

[47] Wikipedia contributors. *Vectored I/O — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Vectored_I/O (visited on 12/30/2020).

[48] Tyler Treat. *Everything You Know About Latency Is Wrong*. Dec. 12, 2015. URL: https://bravenewgeek.com/everything-you-know-about-latency-is-wrong/.

[49] Richard Speed. *Party like it's 2004: Almost a quarter of Windows 10 PCs living with the latest update*. Sept. 1, 2020. URL: https://www.theregister.com/2020/09/01/microsoft_in_brief/.

[50] .NET Platform. *System.Net.Experimental.MsQuic: MsQuic for .NET 5*. URL: https://github.com/dotnet/runtimelab/tree/feature/System.Net.Experimental.MsQuic (visited on 12/11/2020).

[51] Microsoft. *PowerShell Documentation: Official product documentation for PowerShell*. URL: https://docs.microsoft.com/en-us/powershell/ (visited on 12/28/2020).

[52] Cory Nelson. *Proposal: Zero allocation connectionless sockets*. Sept. 9, 2019. URL: https://github.com/dotnet/runtime/issues/30797.

[53] Wikipedia contributors. *TCP offload engine — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/TCP_offload_engine (visited on 12/28/2020).

[54] Yung-Chih Chen et al. "Characterizing 4G and 3G Networks: Supporting Mobility with Multi-Path TCP". In: *UMass Amherst Technical Report: UMCS-2012-022* (Jan. 2012).

[55] Wikipedia contributors. *Burst error — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Burst_error (visited on 12/26/2020).

[56] Injong Rhee et al. *CUBIC for Fast Long-Distance Networks*. Tech. rep. 8312. RFC Editor, Feb. 2018. 18 pp. DOI: 10.17487/RFC8312. URL: https://rfc-editor.org/rfc/rfc8312.txt.

[57] Marten Seemann. *QUIC Interop Test Runner*. URL: https://github.com/marten-seemann/quic-interop-runner (visited on 12/10/2020).

[58] .NET Platform. *.NET Runtime Lab*. URL: https://github.com/dotnet/runtimelab (visited on 12/14/2020).

List of Figures

List of Tables

List of Listings

A. Attachments

Contents of the attachment archive:

```
/
├─ bin ........................ Prepared built binaries
│  ├─ linux-x64 ............... Built binaries for Linux
│  │  ├─ dotnet ............... Patched .NET 6 SDK with managed QUIC implementation
│  │  ├─ msquic ............... MsQuic binaries used for performance comparison in chapter 6
│  │  ├─ openssl .............. Modified OpenSSL binaries with QUIC-enabling API
│  │  └─ ThroughputTests        Self-contained build of benchmarking application from chapter 6
│  └─ win-x64 ................. Built binaries for Windows
│     ├─ dotnet ............... Patched .NET 6 SDK with managed QUIC implementation
│     ├─ msquic ............... MsQuic binaries used for performance comparison in chapter 6
│     ├─ openssl .............. Modified OpenSSL binaries with QUIC-enabling API
│     └─ ThroughputTests        Self-contained build of benchmarking application from chapter 6
├─ certs ...................... Certificates for use in tests and attached samples
│  ├─ cert.crt ................ Public certificate file
│  ├─ cert.key ................ Private key file
│  └─ cert.pfx ................ Certificate and private key combined in PFX format
├─ extern ..................... Third-party source code
│  └─ openssl ................. Copy of the QUIC-enabled fork of OpenSSL by Akamai
├─ src ........................ Root for all source code
│  ├─ dotnet-runtime .......... .NET runtime fork with managed QUIC implementation
│  └─ supplementary ........... Root of additional .NET projects
│     ├─ benchmark ............ Root of all benchmark projects
│     │  └─ ThroughputTests ... Benchmarking application used in chapter 6
│     │     └─ run.ps1 ........ Script used to automate running the benchmarking application
│     ├─ samples .............. Root of all sample projects
│     │  ├─ Echo .............. Simple echo server and client implementation from chapter 5
│     │  └─ hello-net6.0 ...... Hello world application for testing .NET 6 installation
│     └─ ManagedQuic.sln ...... .NET Solution file for all projects in subdirectory
├─ tex ........................ LaTeX source code for this thesis
│  └─ measurements ............ Raw CSV data with measurements presented in chapter 6
│     ├─ linux-lan ............ Measurements in the Linux LAN environment
│     └─ windows-loopback ..... Measurements in the Windows Loopback environment
└─ README.md .................. File describing the contents of the archive
```

Glossary

Abstract Factory Pattern
A design pattern encapsulating creation of families of related classes. A typical example are factories for creating GUI widgets, where different factories can create various types of widgets backed by a particular GUI rendering library. 58

Ack-eliciting Packet
A QUIC packet which contains at least one frame which is not `PADDING`, `ACK`, or `CONNECTION_CLOSE` 22

Application-Layer Protocol Negotiation (ALPN)
A TLS extension that allows the application layer to negotiate which application protocol will be in the connection. The negotiation is done as part of the TLS handshake and avoids additional round trips. ALPN is used, e.g., for HTTP version negotiation in HTTPS connections. 33

Authenticated Encryption With Associated Data (AEAD)
Form of encryption which simultaneously assure the confidentiality and authenticity of data. 34

Connection ID
An opaque identifier that is used to identify a QUIC connection at an endpoint. Each endpoint sets a value for its peer to include in packets sent towards the endpoint. 14

Destination Connection ID (DCID)
Connection ID used by the receiver of the QUIC packet. 17

Garbage Collector (GC)
A part of the .NET runtime which provides automatic memory management. 45

Head-of-line Blocking
Performance limiting phenomenon caused by a line of packets being held up by the first packet 5

Key Update
A process in which QUIC endpoints derive new encryption keys to use for encrypting 1-RTT packets. This process limits the number of packets encrypted by the same encryption key, increasing the level of protection. The process of key update is described in section 2.8.2. 37

Managed Code
Code written in one of the .NET languages running on the .NET virtual machine. 8

Micro-bursting
: A performance limiting phenomenon in which packets arrive in short rapid bursts. These bursts may cause overflow in the receiving buffers and cause the receiver to discard incoming packets. 64

Native Code
: Code written in ahead-of-time compiled language. This code runs directly on the target CPU. 8

Network Path
: An imaginary path between two network addresses. Each end of a network address consists of a pair of local address and port number. 14

Out-of-order Packet
: A packet that does not arrive directly after the packet that was sent before it. A packet can arrive out of order if it is delayed, if earlier packets are lost or delayed, or if the sender intentionally skips a packet number. 13

Packet Pacing
: A mechanism which evens out microbursts of packets in order to prevent network congestion. An ideal network pacer spreads entire congestion window worth of traffic evenly over the round trip time period. 64

Path Validation
: A process during which QUIC endpoint validates that it's peer is reachable via a particular network path. Consists of an exchange of `PATH_CHALLENGE` and `PATH_RESPONSE` frames. 38

QUIC Packet
: A complete processable unit of QUIC that can be encapsulated in a UDP datagram. Multiple QUIC packets can be encapsulated in a single UDP datagram. 13

Server Name Indication (SNI)
: A TLS extension that allows the client to specify the hostname it is attempting to connect to at the start of the handshake process. This allows using different security configurations for each different website hosted on the server. 33

Source Connection ID (SCID)
: Connection ID used by the sender of the QUIC packet. 17

Strategy Pattern
: A behavioral design pattern which allows selecting an internal implementation (algorithm) at runtime. 52

Traffic Amplification Attack
: A type of distributed denial-of-service (DDoS) attack in which attacker sends a small amount of data to a server and the server responds with larger amount of data to the target. An example of such attack is initiating

new connections with a falsified source IP address which make the server flood the victim with handshake attempts. 38

Zero Round Trip Time Resumption (0-RTT)
TLS mode of operation allowing clients to send application data in the very first but possibly exposing the server to reply attacks. 7