A Critical Analysis of QUIC Vs TCP: Within the Context of a Stock Market Order-Matching System

Luke Barry

A Final Year Project

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

BA Joint Honours of Computer Science and Business

Supervisor: Stefan Weber

April 2025

A Critical Analysis of QUIC Vs TCP: Within the Context of a Stock Market Order-Matching System

Luke Barry, BA Joint Honours of Computer Science and Business

University of Dublin, Trinity College, 2025

Supervisor: Stefan Weber

...ABSTRACT... This document is split into two parts: The general guidance towards a structure for a dissertation and information that should facilitate the writing of a dissertation as LaTeX document. The 2nd part of this document has been moved into the appendix - make sure that you read through this part before starting to write your dissertation.

... and before you submit, remove the \thesisdraft tag in the thesis.tex file!!!!

Acknowledgments

Thank you Mum & Dad.

Luke Barry

University of Dublin, Trinity College
April 2025

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This introduction will give the reader a preliminary glance into the material that will be covered in this project. The goal of this project is to illuminate the differences between the Transmission Control Protocol (TCP) and Quick UDP Internet Connections (QUIC), detailing how QUIC improves upon its predecessor, critically analyzing various QUIC implementations and exploring what role QUIC may play in the future of the Internet.

As part of this introduction I will; Discuss my motivation for choosing this research topic, detail my research question that informed my exploration of the academic literature and software development, and finally I aim to devise the structure and contents of the rest of the report.

1.1 Motivation

This section of the report will describe my inspiration and motivation for choosing to explore my chosen subject. The research undertaken as part of this project is relevant for a number of reasons. For instance; QUIC is still in the early stages of its adoption, TCP still remains the dominant means of transporting information across the internet, responsible for more than 85% of network traffic Lee et al. (2010). Moreover, the standards and beneficial behavior of QUIC as laid out by the IETF may in fact differ depending on the QUIC implementation. I first took an interest to Computer Networks in my 2nd year of studies, where I had the opportunity to develop protocols on top of the User Datagram Protocol (UDP). During my studies into Computer Networks I became familiar with both UDP and the Transmission Control Protocol (TCP), particularly how TCP functions and how it controls the reliable and ordered delivery of information across the internet. However, I also learned about the limitations of TCP in areas such as Multiplexing, Connection Migration, Establishment etc. These previous studies allowed me to critique TCP against UDP and by extension its new adversary, QUIC.

The reason for my choice to frame this study in the context of a Stock Market Order-Matching Engine was due to a personal interest of my own. Ever since I took my intern-

ship with Susquehanna International Group LLP - a prominent Proprietary Quantitative Trading Firm - I have been deeply interested with the intersection between Technology & Finance. It was at Susquehanna that I had the opportunity to delve into the development of Finance-Technology solutions, trading and a number of tangential areas. It was accordingly a natural progression of my professional development to choose to opt for an interesting application of the QUIC transport protocol. I personally find practical development to be an effective method of learning, and this influenced my decision to focus much of my efforts on developing this Order-Matching Software as a proof of concept for the applicability of QUIC.

1.2 Research Question

How does QUIC claim to improve data transport relative to TCP, and are these claims consistent with my chosen QUIC implementation, AIOQUIC for Python? Moreover, how does QUIC (aioquic) hold up as means of transport in a Matching Engine compared to existing protocols e.g. SIVA (TCP), FIX/FAST (UDP)?

This research question addresses three fundamental aspects of modern network protocol implementation and evaluation:

First, it examines the theoretical improvements that QUIC promises over TCP, particularly in areas such as connection migration, stream multiplexing, and integrated security. While these improvements are well-documented in specifications, their practical implementations across programming languagesd remains an important area of investigation.

Second, it evaluates the real-world implementation of these features in aioquic, a Python-based QUIC library. This evaluation is crucial because the effectiveness of a protocol is heavily dependent on the quality and completeness of its implementation. By focusing on aioquic, this research provides valuable insights into the challenges and successes of implementing QUIC in a high-level programming language environment such as Python.

Third, it contextualizes QUIC's applicability in the demanding domain of financial trading systems, specifically comparing its performance characteristics against established protocols in matching engine implementations. This comparison is particularly relevant as trading platforms require both the reliability of TCP and the speed of UDP, making them an ideal test case for QUIC's hybrid approach.

The question is structured to yield quantifiable results through:

- Empirical testing of QUIC's key features (connection migration, 0-RTT establishment, stream multiplexing)
- Performance metrics comparison between QUIC and traditional protocols

- Practical evaluation of QUIC's suitability for real-time trading applications

By addressing these aspects, this research contributes to both the theoretical understanding of modern transport protocols and their practical application in performance-critical systems. The findings will be particularly valuable for developers and system architects considering QUIC for similar high-performance applications.

1.3 Structure & Contents

This section will describe the following structure and contents of the proceeding chapters:

Chapter 2 - Transport Protocols

This chapter will focus on providing background into the two transport protocols which are the main focus of my project; the Transmission Control Protocol (TCP) and Quick UDP Internet Connections (QUIC), and by extension the User Datagram Protocol. Furthermore, I will delve deeper into other transfer protocols similar to QUIC which are also built on top of UDP.

Chapter 3 - Packet Capturing/Visualization

This chapter will introduce the software tools that I used to capture and visualize network traffic of my Dockerised testing scenarios and Stock Market Order Matching Software implementation. Moreover, I will demonstrate how I was able to carry out the following captures as well as the process that was required to transform these traffic captures to an easily interpretable visual representation.

Chapter 4 - Docker

This chapter will describe in detail the basic components of how a Dockerised testing environment is established. This aims to clarify to the reader how I was able to create isolated and reproducible testing scenarios. These testing scenarios were carried out in an effort to verify the acclaimed beneficial characteristics of the QUIC transport protocol, as I had read about them in the academic literature I researched as part of this project.

Chapter 5 - State of the Art Research

This chapter will cover more of the technical background of QUIC, the impetus for its development, the reasons said development, etc. All of this background will be framed in a comparative manner to the long-standing and widely adopted TCP. Moreover, as part of my research I will also compare QUIC with a number of transfer protocols which are already utilized for Order Matching software.

Chapter 6 - Problem Formulation

This chapter will provide the reader with an overall description of the problem formulation I have come up with based on the existing work and research covered in the previous chapter. My problem formulation is three-fold; 1. Is the behavior/design of QUIC as laid out in standard documentation by the Internet Engineering Task Force (IETF) consistent with my chosen implementation? (aioquic), 2. Does this observed behavior/design solve the limitations of TCP?, 3. Is QUIC a suitable transport protocol for my chosen Stock Market application?

Chapter 7 - Design

This chapter will discuss the design of both the Dockerised testing scenarios and the Stock Market Order-Matching Engine. This is where I will define the functional requirements of both the testing and the application that I developed as part of this project.

Chapter 8 - Implementation

This chapter will focus on the implementation of the aforementioned testing and application software. It will also illustrate the difficulties and issues I encountered as part of implementation; namely to do with Docker, AIOQUIC, TLS Encryption, SSL Certificates etc.

Chapter 9 - Evaluation

This chapter will focus on evaluating the results obtained from my testing scenarios, comparing the results I collected with the standard behavior as defined in Academic Literature. Moreover, I will also evaluate the suitability of QUIC as a means of transport for a Stock Market application, comparing my application with pre-existing solutions such as FIX/FAST.

Chapter 10 - Conclusion and Future Work

This chapter will focus on my final conclusions surrounding my project and the problems I had initially laid out to address. I will discuss areas of future work that I would pursue (or recommend others to pursue) to further the work I undertook as part of this project.

Chapter 2

Transport Protocols

This chapter focuses on providing a background into the two transport protocols central to this project: the Transmission Control Protocol (TCP) and Quick UDP Internet Connections (QUIC). By extension, the discussion encompasses the User Datagram Protocol (UDP) which QUIC and a number of other protocols are built off of. The latter part of the chapter reviews protocols used particularly for electronic trading.

2.1 Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) was first proposed in May 1974 by Vint Cerf & Bob Kahn and later standardized in September 1981 in Request for Comments (RFC) 793 by the Internet Engineering Task Force (IETF)Postel (1981). It is a connection-oriented protocol that guarantees reliable, in-order delivery of a continuous stream of data between applications. TCP establishes connections using a three-way handshake, ensuring that both sender and receiver are synchronized before any data transfer occurs. It incorporates mechanisms for error detection and recovery, flow control, and congestion control, making it well-suited for applications where data integrity is critical, such as web browsing, email, and file transfers. However, the extensive reliability features of TCP introduce overhead and increased latency, which renders it unsuitable for more latency-sensitive applications.

2.2 User Datagram Protocol (UDP)

The User Datagram Protocol (UDP) was defined in RFC 768 in 1980 by the IETF and represents a significant departure from the connection-oriented model of TCP. As a connectionless protocol, UDP transmits data in the form of datagrams without establishing a persistent end-to-end connection, reducing connection overhead and latency. UDP is ideal for applications that prioritize speed over reliability, such as live video streaming, online gaming, and Voice over IP (VoIP). By leaving error handling, data ordering, and

retransmission responsibilities to the application layer, UDP offers developers the flexibility to implement custom protocols suited to the specific needs of their applications Postel (1980).

2.3 Quick UDP Internet Connections (QUIC)

Quick UDP Internet Connections (QUIC) is a modern transport protocol designed to bridge the gap between the ordered reliability of TCP and the low latency of UDP. Initially developed by Google and later standardized by the IETF in May 2021 Borman et al. (2021), QUIC leverages UDP as its underlying transport mechanism to reduce connection establishment delays. It supports 0-Round-Trip-Time (RTT) and 1-RTT handshakes, allowing data to be transmitted almost immediately without the overhead of traditional TCP connection setups. QUIC makes the design choice of doing away with TCP's client identification 4-tuple (source port number, source IP address, destination port number, destination IP address) and replacing it with Connection Identifiers, this allows QUIC to benefit from resistance to connection migration and related issues. Furthermore, QUIC introduces multiplexing of multiple data streams within a single connection, which mitigates the head-of-line blocking issues that can occur in TCP. Moreover, by integrating robust encryption via Transport Layer Security (TLS) and enhanced congestion control, QUIC provides a level of security and reliability similar to TCP, whilst maintaining the efficient and performant characteristics of UDP. This hybrid approach makes QUIC particularly attractive for applications that require both fast, responsive communications and dependable data delivery.

2.3.1 Connection Establishment Overhead

One of TCP's primary limitations is its connection establishment process, which requires a three-way handshake before any data can be transmitted. This process typically takes 1-2 Round-Trip Times (RTTs) for TCP alone, and when combined with TLS encryption, requires additional RTTs for security negotiation Nepomuceno et al. (2018). This overhead becomes particularly significant in scenarios requiring frequent connection migration/establishment e.g. Mobile Networks or in high-latency networks Kyratzis and Cottis (2022).

QUIC addresses this limitation through several innovative approaches Borman et al. (2021):

1. **0-RTT Connection Establishment**: For repeat connections where the client and server have communicated previously, QUIC can send application data immediately along with the initial handshake, eliminating the traditional connection establishment delay Kumar (2020).

2. **Combined Cryptographic and Transport Handshake**: QUIC integrates the cryptographic (TLS) handshake with connection establishment, reducing the total number of round trips required to 1-RTT Iyengar and Thomson (2021) Thomson and Turner (2021) Kyratzis and Cottis (2022).

3. **Connection Parameters Caching**: QUIC clients can cache server parameters from previous connections, enabling faster reconnection without the need for full parameter negotiation Kosek et al. (2021).

Recent performance evaluations have shown that QUIC's connection establishment latency can be up to 3 times faster than TCP+TLS in typical scenarios Kyratzis and Cottis (2022), making it particularly beneficial for applications requiring frequent connection establishments or operating in high-latency environments.

2.3.2 Connection Migration

TCP connections are bound to a 4-tuple of source IP, source port, destination IP, and destination port. This rigid binding means that if any of these parameters change (e.g., when a mobile device switches from WiFi to cellular networks), the TCP connection must be terminated and re-established Bauer et al. (2023). This limitation becomes increasingly problematic in modern mobile environments where network transitions are common.

QUIC introduces Connection IDs to solve this problem Borman et al. (2021):

1. **Connection ID-Based Routing**: Instead of using IP addresses and ports for connection identification, QUIC employs unique Connection IDs that remain valid across network changes Iyengar and Thomson (2021). Upon connection establishment, the client and server exchange a number of connection IDs.

2. **Seamless Migration**: When network parameters change, QUIC can maintain the existing connection by updating the path verification without requiring a new handshake, reyling on the previously established connection identifiers. It must be noted that all QUIC implementations must enforce that the client switch to a new connection ID if migration occurs, as keeping the same connection ID might lead to privacy concerns around the possibility of tracking a client based off of a single connection ID. Kosek et al. (2021) Borman et al. (2021).

3. **Multi-Path Support**: The Connection ID architecture allows for potential future extensions supporting simultaneous use of multiple network paths. König et al. (2023).

Performance studies have demonstrated that QUIC connections can survive network transitions with minimal interruption, whereas TCP connections require full re-establishment, leading to significant application-level delays Kumar (2020).

2.3.3 TLS Encryption

In traditional network stacks, TCP and TLS operate as separate protocols, with TLS running as an additional layer above TCP. This layered approach has specific implications for connection establishment and security Iyengar and Thomson (2021). When establishing a connection, TCP first requires its three-way handshake to complete, followed by the TLS handshake, resulting in at least two Round-Trip Times (RTTs) before encrypted data can be exchanged Nepomuceno et al. (2018).

QUIC takes a fundamentally different approach by integrating the transport and security layers Borman et al. (2021):

1. **Integrated Encryption**: QUIC mandates the use of TLS 1.3 as an integral part of the protocol, ensuring that cryptographic parameters are negotiated alongside transport parameters in a single handshake Iyengar and Thomson (2021); Kumar (2020).

2. **Reduced Handshake Overhead**: By combining the cryptographic and transport handshakes, QUIC achieves encrypted data exchange in a single RTT for new connections, and zero RTT for resumed connections with cached parameters Kosek et al. (2021) Nepomuceno et al. (2018).

3. **Packet Protection**: QUIC encrypts both the payload and most transport parameters, with studies showing this provides better protection against traffic analysis compared to TCP+TLS Kocak et al. (2022).

4. **Key Updates**: QUIC supports in-connection key updates without requiring additional handshakes, allowing for regular key rotation in long-lived connections Iyengar and Thomson (2021).

Performance analyses have demonstrated that QUIC's integrated security approach reduces connection establishment time by up to 50% compared to separate TCP and TLS handshakes **?**, while maintaining equivalent security guarantees Kocak et al. (2022).

2.3.4 Multiple Streams & solving the Head-Of-Line-Blocking problem

TCP's ordered delivery leads to problems in circumstances where multiple streams of data are being transmitted at once. If this is to be avoided, the client must establish numerous connections with the server, this comes with additional establishment overhead and bandwith usage. A lost packet on a single stream blocks all subsequent data on all other streams until it is retransmitted, even if the subsequent data belongs to independent application streams Nepomuceno et al. (2018). This is known as the head-of-line blocking

(HOLB) problem. While there is a single TCP-based protocol; SPDY, which supports multiplexing, it is not supported by many networks and thus its interoperability with the wider internet is questionable. This limitation significantly impacts performance in scenarios requiring multiple parallel data transfers Kumar (2020).

QUIC addresses this through native stream multiplexing, facilitating numerous streams each capable of transporting their own data independently from one another. Borman et al. (2021):

1. **Independent Streams**: QUIC allows multiple logical streams within a single connection, each with independent flow control and error handling Iyengar and Thomson (2021).

2. **Stream-Level Flow Control**: Each stream has its own flow control mechanisms, preventing a slow stream from impacting others Kumar (2020).

3. **Prioritization**: QUIC supports stream prioritization, allowing applications to optimize resource allocation based on their specific needs Kosek et al. (2021).

4. **Reduced Latency**: By eliminating HOL blocking between streams, QUIC significantly reduces latency for multiplexed applications.

Performance evaluations have shown that QUIC's stream multiplexing can improve throughput by up to 30% compared to TCP in scenarios with multiple parallel data transfers König et al. (2023). This is particularly relevant for modern web applications that typically require multiple simultaneous resource transfers.

2.4 Protocols used for Electronic Trading

Electronic Trading Systems - such as the ones used for Stock Market Order-Matching Software - rely on specialized protocols to efficiently transmit different types of financial data. These protocols are built on top of the transport layer protocols discussed earlier and are optimized for the unique requirements of the financial markets they facilitate.

2.4.1 Order Flow

Order flow refers to the transmission of trade instructions from market participants to execution venues e.g. Stock Exchanges. These instructions include order creation, modification, and cancellation requests sent by traders, brokers, and algorithmic trading systems to exchanges or other market venues.

Order flow data has several distinct characteristics that influence protocol selection:

1. **Low latency requirements**: Order execution speed can significantly impact trading outcomes, particularly in high-frequency trading where microseconds matter. The time taken for an order to reach an exchange and receive confirmation (round-trip time) is critical.

2. **Reliability requirements**: Orders represent financial commitments, making guaranteed delivery essential. Lost or corrupted orders can result in significant financial losses or missed opportunities.

3. **Transaction integrity**: Orders must be processed exactly once, in the correct sequence, with no duplication or loss.

4. **Moderate bandwidth usage**: Individual orders are typically small in size compared to market data feeds, but trading venues systems need to accommodate thousands of orders per second.

These requirements have made TCP-based protocols the preferred choice for order flow, as the guaranteed delivery and in-order processing align well with order transmission requirements, despite the additional latency from connection establishment and acknowledgment mechanisms.

2.4.2 Market Data

Market data consists of real-time information about financial instruments being traded in the markets, including:

1. **Quote data**: Bid and ask prices and quantities at which market participants are willing to transact.

2. **Trade data**: Records of executed transactions including price, volume, and timestamp.

3. **Order book data**: The full set of resting orders at various price levels.

4. **Reference data**: Information about the traded instruments, corporate actions, and other market events.

Accordingly, Market data has distinct characteristics from order flow:

1. **High volume**: Market data can reach millions of messages per second during peak trading periods.

2. **Distribution pattern**: Market data follows a one-to-many distribution model, where exchanges broadcast the same data to numerous subscribers.

3. **Timeliness over completeness**: For many applications, it is better to receive recent market data promptly than to wait for retransmission of missed messages.

4. **Varied latency sensitivity**: While some applications (like high-frequency trading) require minimum latency, others prioritize completeness and may tolerate some delay.

Due to these characteristics, market data distribution often employs UDP-based multicast protocols that sacrifice guaranteed delivery for lower latency and higher throughput. Modern systems increasingly explore UDP-based solutions to balance reliability with performance.

2.4.3 FIX/FAST

The Financial Information eXchange (FIX) protocol is the industry standard for communicating trade information in financial markets. Developed in 1992 and maintained by FIX Trading Community, it provides a common language for pre-trade, trade, and post-trade communications.

FIX Protocol Structure and Characteristics

1. **Message format**: FIX messages consist of tag-value pairs separated by the ASCII SOH character (represented as '—' in text). Each message includes a header, body, and trailer section.

2. **Session layer**: FIX includes a session layer for managing connections, providing authentication, sequence numbering, and heartbeat functionality.

3. **Transport independence**: While FIX is most commonly implemented over TCP to ensure reliable delivery, the protocol itself is transport-agnostic.

4. **Versions**: The protocol has evolved through multiple versions (FIX 4.0, 4.2, 4.4, 5.0, etc.), each expanding the message types and fields to accommodate new trading requirements.

As financial markets evolved and the volume of market data increased, the verbosity of the traditional FIX format became a limitation. This led to the development of FIX Adapted for Streaming (FAST), which provides efficient message encoding and compression:

FAST Protocol Enhancements

1. **Field encoding**: FAST uses a template-based approach to encode fields, drastically reducing message size compared to standard FIX.

2. **Bandwidth optimization**: FAST transmits only changes in field values rather than complete messages, using techniques like delta encoding and presence maps.

3. **UDP compatibility**: While standard FIX typically runs over TCP, FAST is often implemented over UDP for market data distribution, leveraging multicast capabilities for efficient one-to-many transmission.

4. **Relationship to transport protocols**: FAST's efficiency makes it particularly suitable for high-volume market data distribution over UDP, while traditional FIX remains predominantly TCP-based for order flow. QUIC's capabilities offer potential advantages for both, combining reliable delivery with improved performance.

2.4.4 Kafka

Apache Kafka has gained significant adoption in electronic trading systems as a distributed streaming platform. Originally developed by LinkedIn and later open-sourced, Kafka serves as a high-throughput, fault-tolerant messaging system.

Kafka Architecture and Features

1. **Publish-subscribe model**: Kafka implements a distributed publish-subscribe messaging system where publishers (producers) send messages to topics, and subscribers (consumers) reads messages from those topics.

2. **Persistent storage**: Unlike traditional message queues, Kafka stores messages on disk with configurable retention periods, allowing consumers to replay historical data.

3. **Partitioning**: Topics are divided into partitions, enabling parallel processing and horizontal scalability.

4. **Consumer groups**: Multiple consumers can form groups to collectively process data from different partitions of the same topic.

5. **Low latency**: While not as low-latency as specialized market data protocols, Kafka offers millisecond-level performance suitable for many trading applications.

Kafka in Trading Environments

1. **Market data distribution**: Kafka can serve as a central distribution point for market data, with exchanges or data providers publishing to topics and various applications consuming according to their needs.

2. **Order flow integration**: Trading firms use Kafka to integrate various order management systems, risk checks, and execution algorithms. E.g. FIX Drop Copy Trade Reconciliation.

3. **Transport protocol relationship**: Kafka relies on TCP for its internal communications, prioritizing reliability and exactly-once delivery semantics. This differs from UDP-based market data protocols but aligns with Kafka's design goal of guaranteed message delivery.

When it comes to electronic trading applications, protocol selection involves balancing the competing demands of latency, reliability, and throughput. While traditional systems have relied on TCP for order flow and UDP for market data, the emergence of QUIC presents an opportunity to consolidate these functions under a single transport protocol that combines the advantages of both approaches.

Chapter 3

Packet Capturing & Analysis

3.1 TCPDump

TCPDump is a powerful command-line packet analyzer that allows for capturing network traffic directly from the network interface. In my Dockerized testing environments, I used TCPDump to capture all QUIC traffic between the client and server components.

The implementation of TCPDump in my Docker environment is shown in the referenced code listing A.1.

This configuration creates a dedicated TCPDump service that:

- Uses the `nicolaka/netshoot` image, which includes network troubleshooting tools
- Captures all UDP traffic on port 8080 (where my QUIC server listens)
- Writes the captured packets to a file named after the relevant testing scenario.
- Shares the network namespace with the server service to capture all its traffic

This approach allowed me to capture all QUIC traffic without modifying my application code, providing a clean separation between the application performance and the monitoring infrastructure.

3.2 WireShark

WireShark is a widely-used network traffic analyzer that allows for the interactive browsing of traffic on a computer network. Created by Gerald Combs in 1997, Wireshark provides a graphical user interface for the detailed inspection and analysis of .pcap files Combs (1997). For my project, WireShark was used to observe the traffic captured during testing, allowing me to inspect the QUIC packets exchanged between the client and server components of my Stock Market Order Matching Software.

WireShark's ability to decode QUIC packets was particularly valuable, as it allowed me to examine the structure and content of the encrypted traffic, providing insights into the performance and behavior of my implementation. This was especially useful at the early stages of development where I was still getting to grips with QUIC.

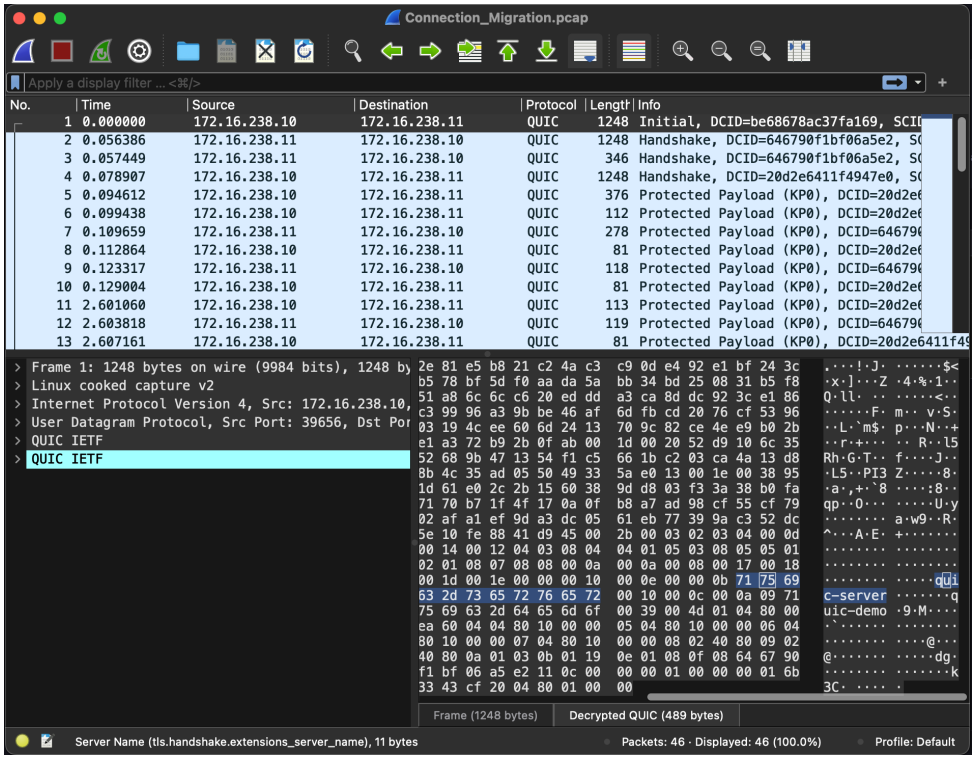


Figure 3.1: Wireshark interface showing QUIC packet analysis

3.3 QLog & QVis

3.3.1 Qlog

Qlog is a standardized logging format specifically designed for QUIC protocol events. It provides a structured way to capture detailed information about QUIC connections, including handshakes, stream management, congestion control, and packet transmission. This logging format was developed as part of the QUIC standardization efforts to help developers debug and analyze QUIC implementations.

In my implementation (aioquic), I utilized the 'QuicFileLogger' class from the aioquic library to automatically generate .qlog files. This logger captures all relevant QUIC events throughout the lifecycle of a connection and writes them to a standardized .qlog file format. The implementation in my client code is shown in the code listing A.5.

The 'QuicFileLogger' is initialized with a directory path where it automatically creates a new .qlog file for each connection with a unique identifier. These log files contain detailed information about:

- Connection establishment and handshake processes
- Stream creation and management
- Packet transmission, including packet numbers and sizes
- Acknowledgment handling and packet loss detection
- Flow control and congestion control mechanisms
- Connection closure events

The logging happens automatically in the background as the QUIC connection operates, requiring no additional code beyond the initial setup. This non-intrusive approach ensures that the logging process doesn't interfere with the normal operation of the application.

3.3.2 QVis

QVis is an open-source visualization tool created by Robin Marx specifically designed for QUIC protocol analysis Marx (2021). It provides a web-based interface for interpreting and visualizing the data contained in .qlog files, making it easier to understand the complex interactions within QUIC connections.

To visualize the Qlog files generated by my application, I followed these steps:

1. Collected the .qlog files from the `./qlogs` directory after test runs
2. Visited the QVis web application at `https://qvis.quictools.info`
3. Uploaded the .qlog files through the web interface
4. Used the various visualization tools provided by QVis to analyze the QUIC connections

QVis offers several powerful visualization views that help interpret the QUIC protocol behavior:

- Sequence diagrams showing packet exchanges between client and server
- Stream timelines displaying data flow on each individual stream
- Congestion control visualizations showing window size changes over time
- RTT (Round Trip Time) measurements and their variation
- Packet loss and recovery information with detailed timing

These visualizations were instrumental in understanding the behavior of my multi-stream QUIC implementation and identifying performance bottlenecks, especially in the context of my Stock Market Order Matching Software where multiple streams handle different order types simultaneously.

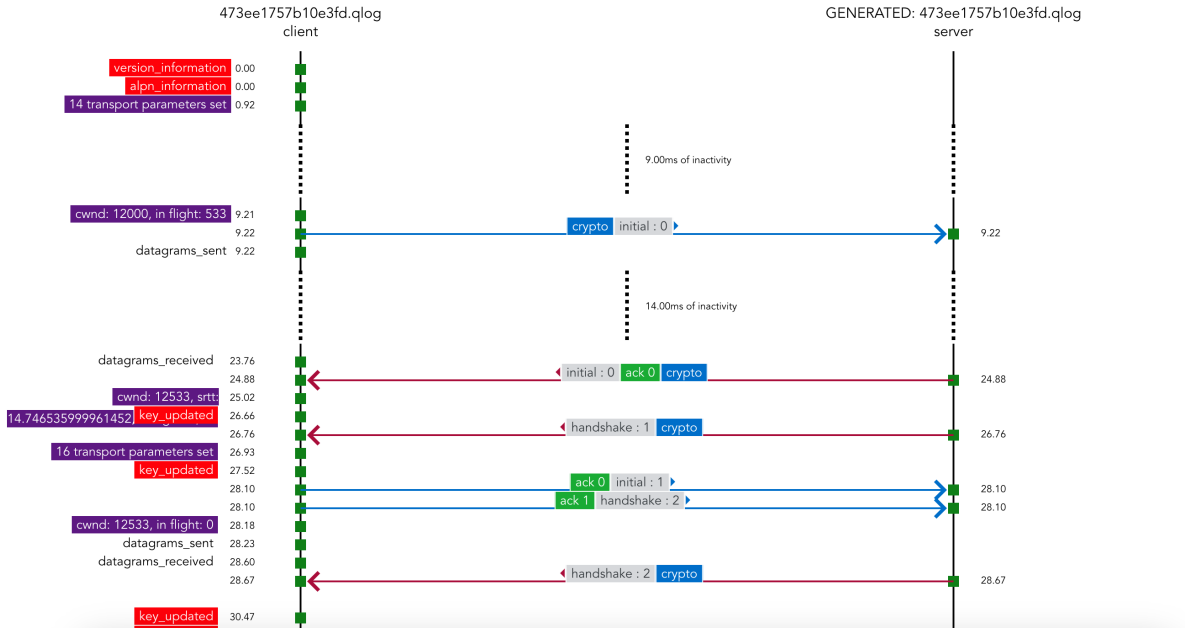


Figure 3.2: Example QVis visualization of the sequence of communication between Client and Server

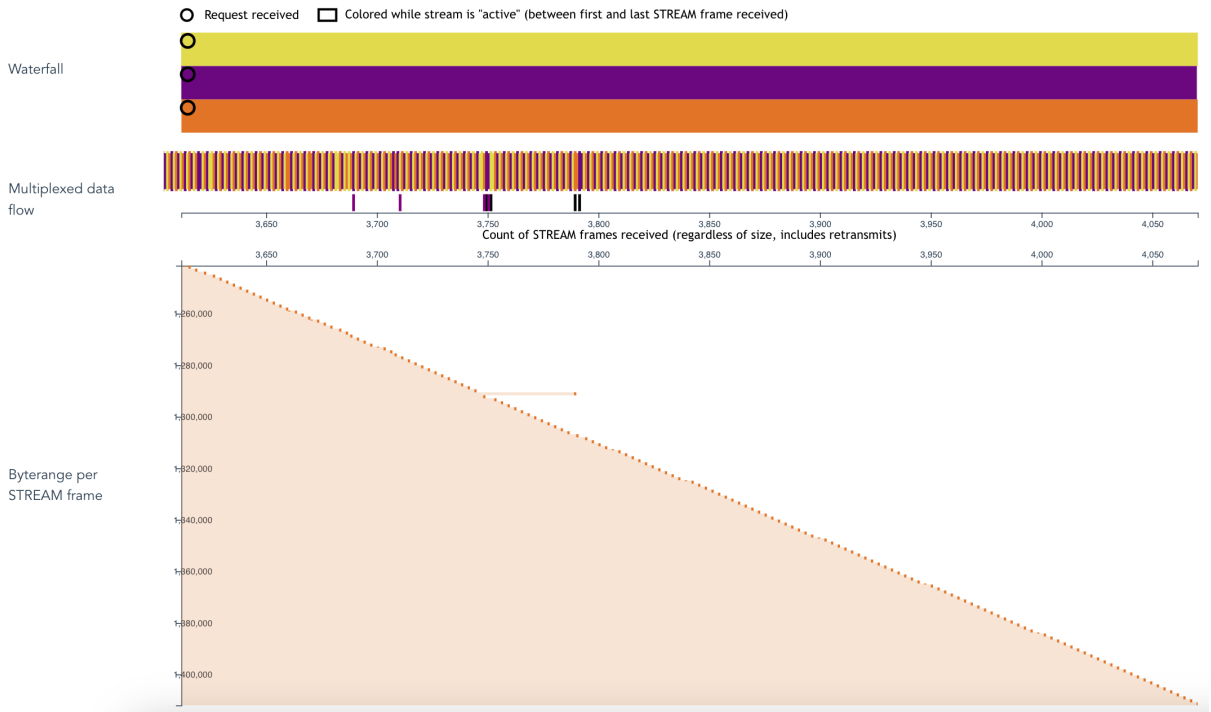


Figure 3.3: Example QVis visualization of distribution of multiplexed communication between Client and Server. N.B. Each colour is a separate stream.

Chapter 4

Docker

4.1 Containers & Images

Docker containers provide a lightweight, standalone, and executable software package that includes everything needed to run an application: code, runtime, system tools, libraries, and settings. For my QUIC protocol testing scenarios, Docker containers were essential in creating isolated testing environments that ensured consistent and reproducible results.

Each testing scenario in my project utilized custom Docker images built from Dockerfiles. These Dockerfiles defined the environment configuration required for both the QUIC client and server components for testing. As shown in A.2, the use of multi-stage builds optimized the final image size while ensuring all dependencies were properly installed.

The Dockerfiles demonstrate several important Docker best practices:

- **Multi-stage builds**: As demonstrated in A.2, the build process is split into two stages - a builder stage that compiles dependencies and a final stage that only includes what's necessary for runtime, resulting in smaller images.
- **Layer caching**: Requirements are copied and installed before the application code to take advantage of Docker's layer caching mechanism, as shown by the BUILDKIT_INLINE_CACHE argument in A.3. This was instrumental in speeding up build times.
- **Minimal base image**: Using python:3.10-slim as the base image reduces the overall size compared to full-featured alternatives.
- **Cleanup**: The removal of unnecessary files after installation is demonstrated in A.2 with the cleanup of wheels and apt lists.

4.2 Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. For my QUIC testing scenarios, Docker Compose was instrumental in orchestrating the complex testing environments, which typically consisted of a client container, a server container, and a network packet capture (TCPDump) container, as shown in A.1.

The Docker Compose configuration defined:

- The services (containers) required for each test, as demonstrated in A.3
- The network configuration for communication between containers
- Volume mounts for sharing data and saving packet capture of communications between the containers
- Resource limits to ensure consistent performance
- Dependencies between services

The configurations demonstrate several key aspects of the testing environment:

- **Custom network**: A dedicated bridge network with fixed IP addresses ensures consistent network conditions across test runs, as shown in both A.3 and A.4.
- **Resource constraints**: Memory limits are applied to ensure consistent performance characteristics, exemplified in A.3 with the 1GB memory limit.
- **Volume mounts**: Certificates and log files are shared between the host and containers for analysis, as demonstrated in the volume configurations across all compose files.
- **Service dependencies**: The client waits for the server to start before attempting to connect, implemented using the depends_on directive in A.4.

Each testing scenario required specific Docker Compose configurations to facilitate its unique requirements. For instance, the Multiple Streams testing environment (A.3) includes environment variables for stream count configuration. This allowed me to conduct variable testing to compare the effects of stream count and packet size, which I will discuss later on. The Connection Migration setup (A.4) focuses on network interface management and packet capture from the client's perspective.

4.3 Testing Scenario Implementations

The Docker configurations were tailored for each testing scenario's specific requirements. Each scenario required unique container setups and networking configurations to effectively test different QUIC protocol features.

4.3.1 Connection Migration Testing

The Connection Migration environment, as shown in A.4, has several unique characteristics:

- TCPDump container uses "network_mode: container:quic-client" to capture packets from the client's perspective
- Network configuration allows for dynamic IP address changes during testing
- Packet captures are stored in a separate capture directory structure
- No specific resource limits, as the focus is on network interface changes rather than performance

4.3.2 Connection Establishment Testing

The Connection Establishment environment focuses on session resumption and requires:

- Persistent volume mounts for session ticket storage
- SSL certificate sharing between client and server
- No specific performance constraints, as the focus is on handshake behavior
- TCPDump configuration optimized for capturing initial connection packets

4.3.3 Multiple Streams Testing

The Multiple Streams setup, detailed in A.3, is unique in its:

- Explicit memory limits (1GB) to ensure consistent performance across stream tests
- Environment variable configuration for dynamic stream count adjustment
- Higher resource reservations for handling multiple concurrent streams
- TCPDump configuration focused on capturing inter-stream communication

4.3.4 Embedded TLS Testing

The TLS encryption testing environment requires specific configurations:

- Shared certificate and key volumes between containers
- SSL keylog file mounting for Wireshark analysis
- TCPDump capture focused on encrypted traffic patterns

- Additional security-related volume mounts for storing SSL-related files

Each testing scenario's Docker configuration was designed to isolate the specific QUIC feature being tested while maintaining consistent environmental conditions for reproducible results. The use of Docker Compose allowed for easy orchestration of these complex testing environments, with each scenario's unique requirements being handled through specific service configurations and volume mappings.

Chapter 5

State of the Art

This chapter presents a comprehensive review of current research and technological developments in transport protocols, with a specific focus on QUIC and its comparison to traditional TCP/UDP implementations. The chapter is structured to provide the reader with both a fundamental understanding and detailed analysis of cutting-edge developments in this field.

The Background section will establish the historical and basic concepts of transport protocols, setting the foundation for understanding current innovations. The Closely-Related Work section then examines three critical aspects of modern transport protocol research: performance comparisons between QUIC and TCP, security implications of QUIC's design, and real-world deployment challenges and solutions.

It is my goal to systematically evaluate the current state of research into the QUIC protocol while maintaining focus on aspects most relevant to stock market order matching systems and high-performance networking applications.

5.1 Background

The evolution of transport protocols has been driven by the changing demands of internet applications and services. While TCP has served as the backbone of internet communications since its standardization in 1981 Postel (1981), modern applications require features beyond its original design parameters. This section provides essential context for understanding current research directions in transport protocol development.

The fundamental shift from TCP to QUIC represents more than just a protocol upgrade; it reflects a broader trend toward integrated, secure, and performance-oriented transport solutions. As detailed in RFC 9000 Borman et al. (2021), QUIC combines the reliability mechanisms of TCP with modern security requirements and performance optimizations.

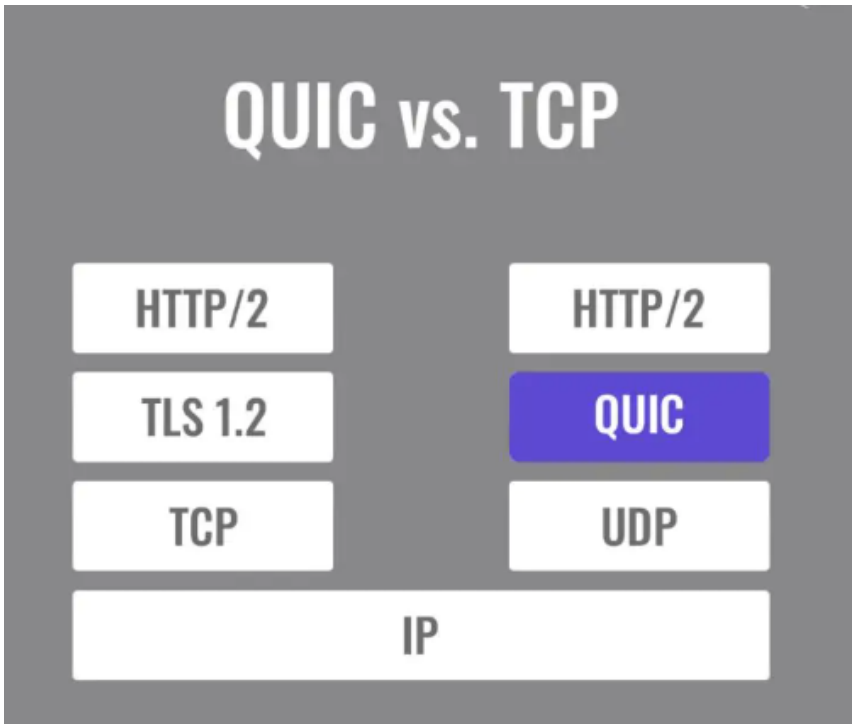


Figure 5.1: Protocol Stack Comparison: Traditional TCP+TLS vs. QUIC

5.2 Closely-Related Work

Recent research in transport protocols has focused on three primary aspects: performance characteristics, security integration, and deployment considerations. Each of these aspects has been extensively studied in the context of QUIC's potential advantages over traditional protocols such as TCP.

5.2.1 Performance Characteristics

Performance evaluation of QUIC compared to TCP has been a major focus of recent research. Kyratzis and Cottis (2022) conducted extensive testing over mobile LTE networks using NS-3 simulations, revealing several key findings:

- QUIC achieved 15-20% lower connection establishment times compared to TCP+TLS
- In lossy networks (scenarios with high rates of packet loss), QUIC's stream multiplexing provided up to 30% better throughput
- 0-RTT connection resumption in QUIC reduced latency by up to 50% for repeat connections

These findings were further supported by Nepomuceno et al. (2018), who conducted real-world experiments comparing QUIC and TCP performance across different network conditions. Their research demonstrated that QUIC's advantages become particularly

pronounced in challenging network conditions, such as those with high packet loss or variable latency. Examples of networks with such conditions includes Internet-of-Things (IoT), e.g. Smart Home Appliances, Connected Retail etc.

5.2.2 Security Integration

The security aspects of QUIC have been thoroughly analyzed in recent literature. Thomson and Turner (2021) provided the foundational specification for QUIC's security architecture, while Kocak et al. Kocak et al. (2022) conducted a comprehensive security analysis comparing QUIC and TCP+TLS implementations.

Key findings from security research include:

- QUIC's integrated security approach reduces the attack surface compared to separate TCP+TLS implementations
- Mandatory encryption of transport parameters provides enhanced privacy
- The combination of transport and security handshakes reduces the opportunity for man-in-the-middle attacks

5.2.3 Deployment Considerations

Recent studies have focused on the practical aspects of QUIC deployment in production environments. Bauer et al. Bauer et al. (2023) conducted extensive research on QUIC's performance in Content Delivery Networks (CDNs) and high-throughput applications, revealing:

- QUIC's connection migration capabilities provide significant advantages in mobile environments
- Implementation complexity can be a barrier to adoption in specialized systems
- Performance benefits vary significantly based on network conditions and application requirements

König et al. König et al. (2023) further explored QUIC's long-term performance characteristics, providing valuable insights into sustained throughput and stability in production deployments.

5.3 QUIC Implementations

While the QUIC protocol is standardized through RFC 9000 Borman et al. (2021), various implementations have emerged with different design philosophies and performance

characteristics. Marx et al. Marx et al. (2020) conducted a comprehensive study of QUIC implementation diversity, revealing significant variations in performance and behavior across different implementations. This section examines three prominent QUIC implementations: aioquic, quiche, and MsQuic.

5.3.1 AIOQUIC

Aioquic, the implementation chosen for this project, is a Python-based QUIC library that emphasizes clean architecture and ease of integration. Performance analysis by Halme Halme (2021) revealed several key characteristics:

- Achieves 85-90% of TCP throughput in ideal network conditions
- Shows consistent performance in handling multiple streams
- Demonstrates excellent stability in long-running connections
- Provides comprehensive logging capabilities through the Qlog format

However, Zirngibl et al. Zirngibl et al. (2022) identified some limitations:

- Higher CPU utilization compared to C-based implementations
- Performance degradation with increasing concurrent connections
- Limited congestion control customization options

5.3.2 Cloudflare's Quiche

Quiche, developed by Cloudflare, is a Rust-based implementation designed for production environments. Wang et al. (2024) conducted extensive performance evaluations showing that quiche excels in several areas:

- Achieves up to 95% of theoretical maximum throughput on high-speed links
- Demonstrates superior memory efficiency under load
- High-performance in handling connection migration
- Provides robust congestion control implementations

Marx et al. (2020) noted that quiche's implementation choices particularly benefit real-world deployment scenarios:

- Aggressive congestion control recovery
- Efficient stream prioritization
- Optimized packet pacing algorithms

5.3.3 Microsoft's MsQuic

MsQuic, Microsoft's C-based implementation, shows distinctive performance characteristics according to Wang et al. (2024):

- Achieves the highest throughput among tested implementations on Windows platforms
- Demonstrates superior CPU efficiency in multi-core scenarios
- Shows excellent scalability with increasing connection counts

However, Halme Halme (2021) identified certain trade-offs:

- Higher memory usage per connection compared to quiche
- More complex configuration requirements
- Platform-specific performance variations

5.3.4 Implementation Comparison

Based on comprehensive benchmarks by Wang et al. Wang et al. (2024), the following table quantifies key performance metrics:

Metric	**aioquic**	**quiche**	**MsQuic**
Max Throughput (Gbps)	2.8	3.2	3.5
CPU Usage (rel.)	1.3x	1.0x	0.9x
Memory/Conn (MB)	0.5	0.3	0.7
0-RTT Success (%)	98	99	99

Table 5.1: Performance Comparison of QUIC Implementations

For this project, aioquic was selected despite not having the highest raw performance metrics. Part of this decision was influenced by my own liking of Python. This decision was also influenced by Marx et al. (2020), who emphasize that implementation choice should be based on use-case requirements rather than just performance metrics. The key factors favoring aioquic for this research project include:

- Superior debugging and logging capabilities, essential for research and testing purposes
- Excellent integration with Python's 'async' ecosystem
- Sufficient performance characteristics for my Stock Market Order-matching use case

- Lower development complexity compared to C or Rust implementations, speeding up development-related hurdles

Zirngibl et al. (2022) note that while aioquic may not match the raw performance of quiche or MsQuic in high-throughput scenarios, its architecture makes it particularly suitable for research and prototyping applications where observability and ease of modification are prioritized over absolute performance.

5.4 Summary

The current state of research demonstrates that QUIC represents a significant advancement in transport protocol design, particularly in areas critical to modern applications such as stock market order matching systems. The following table summarizes the key findings from recent research:

Aspect	**TCP+TLS**	**QUIC**
Connection Establishment	2-3 RTTs	0-1 RTTs
Security Integration	Separate Layer	Integrated
Stream Multiplexing	Not Available	Native Support
Connection Migration	Not Supported	Supported

Table 5.2: Comparison of TCP+TLS and QUIC Features

While QUIC shows clear advantages in many scenarios, the research also highlights important considerations for implementation and deployment. The protocol's complexity and relative novelty present challenges that must be carefully evaluated in the context of specific application requirements.

This review of current research provides a foundation for understanding both the potential benefits and challenges of implementing QUIC in high-performance networking applications, particularly in the context of stock market order matching systems where low latency and reliability are critical requirements.

Chapter 6

Problem Formulation

The evolution of internet transport protocols has been marked by continuous adaptation to changing network conditions and application requirements. While TCP has served as the backbone of internet communications for decades, modern applications, particularly those requiring low latency and high reliability like stock market trading systems, demand more sophisticated transport layer solutions. This chapter will identify the challenges from existing research and specifies the particular problems this dissertation addresses.

6.1 Identified Challenges

Based on the comprehensive review of Academic writing covered in my State of the Art Research, several critical challenges emerged in the context of modern network applications:

6.1.1 Implementation Diversity and Standard Compliance

Marx et al. (2020) highlight a significant challenge in the QUIC ecosystem: the diversity of implementations leads to varying behaviors and performance characteristics. Their research reveals that different QUIC implementations can exhibit up to 30% variation in performance metrics under identical network conditions. This raises questions about not only the consistency of QUIC's benefits across different implementations, but their interoperability with one another and the wider internet.

6.1.2 Performance Trade-offs

Wang et al. (2024) identify specific performance challenges in QUIC implementations:

- CPU overhead in Python-based implementations like aioquic
- Memory utilization variations across different implementations

- Throughput limitations in certain network conditions

6.1.3 Financial Market Requirements

The application of QUIC to financial trading systems presents unique challenges:

- Ultra-low latency requirements for order matching
- Need for reliable multistream communication
- Security requirements without compromising performance
- Connection migration capabilities for failover scenarios

6.2 Proposed Work

This project aims to address these challenges through a three-fold investigation:

6.2.1 Implementation Analysis

The first component involves a systematic evaluation of the aioquic implementation against IETF standards. This analysis will:

- Compare aioquic's behavior with IETF specifications Borman et al. (2021)
- Evaluate performance characteristics in controlled (dockerised) environments
- Assess the impact of Python's runtime on QUIC performance

6.2.2 TCP Limitation Resolution

The second component examines how effectively QUIC addresses TCP's limitations of the aforementioned behaviour:

- Connection establishment overhead
- Head-of-line blocking in multiplexed streams
- Connection migration capabilities
- TLS integration and performance

Based on Zirngibl et al. (2022) findings, special attention will be paid to scenarios where QUIC's theoretical advantages may be affected by implementation choices.

6.2.3 Stock Market Application Evaluation

The third component involves developing and evaluating a stock market order-matching system using QUIC. This practical implementation will:

- Compare QUIC performance against existing protocols (FIX/FAST)
- Evaluate multistream capabilities for order processing
- Assess connection migration for failover scenarios
- Measure the impact of TLS integration on trading latency

6.3 Research Questions

Based on these challenges and proposed work, this dissertation addresses three primary research questions:

1. To what extent does the aioquic implementation align with IETF QUIC standards in terms of both behavior and performance?
2. How effectively does QUIC, specifically the aioquic implementation, address the known limitations of TCP in real-world scenarios?
3. Can QUIC serve as a viable transport protocol for high-frequency trading systems, particularly in comparison to established protocols like FIX/FAST?

These questions will be systematically addressed through both theoretical analysis and practical implementation, with results quantified through comprehensive performance metrics and behavioral analysis. The evaluation will focus not just on raw performance numbers but also on the practical implications for real-world financial systems.

6.4 Expected Contributions

This work aims to make several key contributions to the field:

- A comprehensive analysis of my chosen QUIC implementation (AIOQUIC), its behavior in financial applications
- Practical insights into the challenges and benefits of using AIOQUIC, as well as any discrepancies between AIOQUIC and the QUIC standards as defined in Borman et al. (2021)
- A framework for evaluating transport protocols in the context of stock market systems

- Empirical data comparing QUIC's performance with traditional financial protocols

These contributions will provide valuable insights for both academic research and practical implementations in the financial technology sector.

Chapter 7

Implementation

This chapter details the practical implementation of both the QUIC testing scenarios and the stock market order-matching system which form the practical aspects of my project. The implementation uses the aforementioned containerized environments for testing and a microservices architecture for the trading platform, with QUIC serving as the primary transport protocol.

7.1 QUIC Testing Scenarios

7.1.1 Connection Migration Implementation

The connection migration test validates QUIC's ability to maintain connection state across network changes. Implementation details:

- **Network Migration**: Uses Docker's ability to detach and switch the IP address of containers during runtime. This allows the testing scenario to accurately replicate the same sort of connection migration that might occur when using TCP over a network prone to migration e.g. Mobile Networks.
- **Connection ID Monitoring**: Through the use of TCPDump and Wireshark, I am able to monitor whether connection Identifiers change once migration occurs.
- **Traffic Capture**: Implements continuous message sending to demonstrate connection persistence during IP changes, with network traffic captured via TCPDump for analysis.

7.1.2 Connection Establishment Implementation

The connection establishment test validates QUIC's 0-RTT and 1-RTT handshake capabilities:

- **Session Ticket Management**: Implements JSON-based session ticket storage and retrieval for connection resumption:
  - The client saves session tickets to disk using a structured JSON format that preserves all critical fields including ticket data, cipher suite, validity timestamps, and resumption secret
  - A `session_ticket_handler` callback captures new tickets during connection
  - `load_saved_session_ticket()` validates and loads previously saved tickets, checking expiration times before use
- **Two-Phase Testing**:
  - Phase 1: Initial connection using 1-RTT handshake, establishing a fresh TLS session
  - Phase 2: Connection resumption attempting 0-RTT using the saved session ticket:
    * The client configures early data options with `configuration.enable_early_data = True`
    * The saved session ticket is loaded into `configuration.session_ticket`
    * The cipher suite from the saved ticket is explicitly set for the resumed connection
- **Performance Monitoring**: SSL key logging for Wireshark & QLog analysis of the handshake processes
  - SSL keys are written to a designated file path through the `secrets_log_file` parameter in the QUIC configuration
  - Environment variable `SSLKEYLOGFILE` is set to enable decryption of captured traffic
  - Detailed event logging captures handshake completion events with session resumption status

7.1.3 Multiple Streams Implementation

Demonstrates QUIC's stream multiplexing capabilities with comprehensive metrics collection:

- **Stream Management**:
  - Extended implementation of stream creation for AIOQUIC with locking mechanisms

  - Support for up to 100 concurrent bidirectional streams
  - Patched stream ID management for client-initiated streams

- **Performance Testing**:
  - Variable message sizes: 1KB, 4KB, 16KB, and maximum QUIC packet size
  - Variable stream multiplicity: 1, 3, 5 streams to compare throughput allocation across numerous streams
  - Concurrent throughput testing across all streams
  - Five-second test duration per message size

- **Metrics Collection**:
  - Per-stream throughput measurements
  - Aggregate throughput calculation in Mbps
  - Messages per second tracking
  - JSON-based results storage

7.1.4 Embedded TLS Implementation

Demonstrates QUIC's integrated TLS 1.3 encryption capabilities:

- **SSL Key Logging**:
  - Automatic key logging to '/app/certs/ssl_keylog.txt'
  - Integration with Wireshark for encrypted traffic analysis

- **Security Configuration**:
  - ALPN protocol specification for QUIC
  - Maximum datagram frame size configuration
  - Self-signed certificate handling

- **Message Exchange**:
  - Stream-based encrypted communication
  - Event-driven message handling
  - Asynchronous protocol implementation

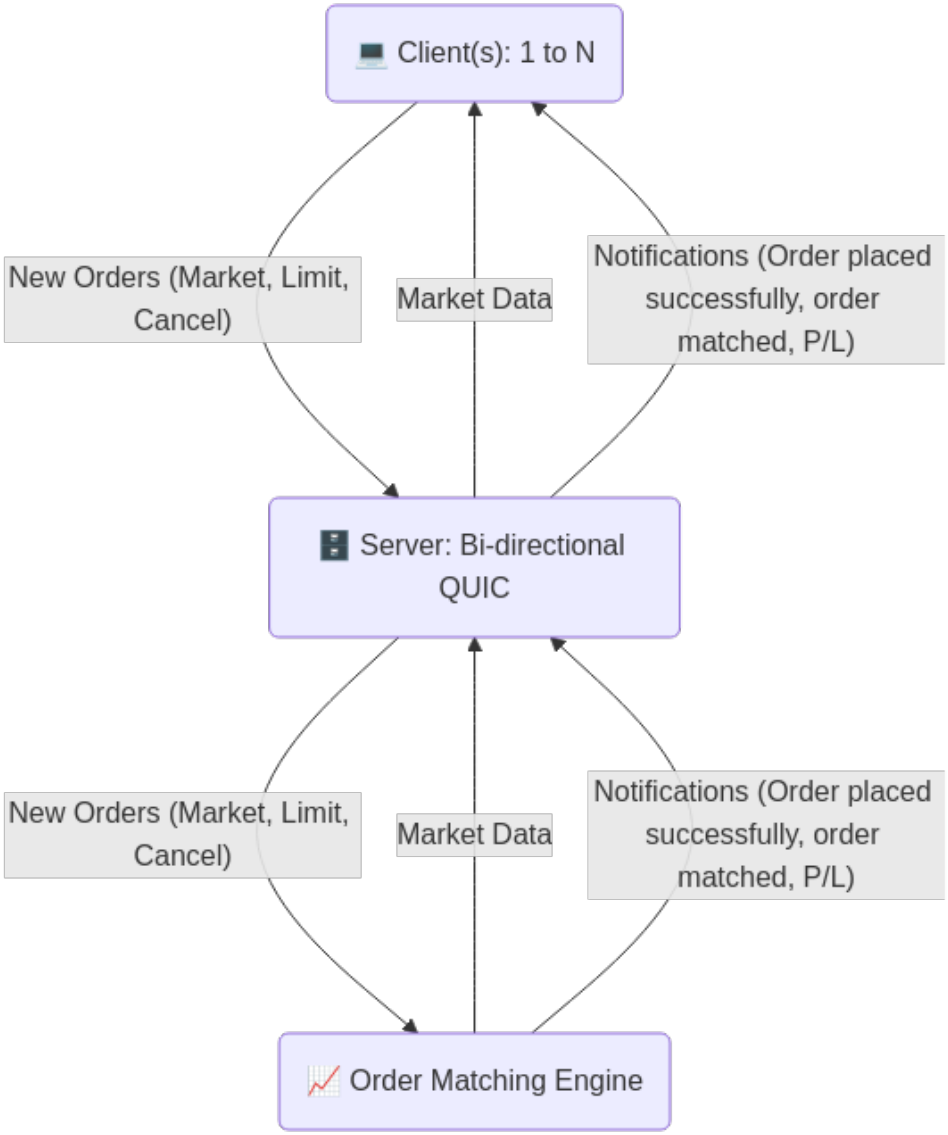
Each testing scenario is containerized using Docker, with specific network configurations and traffic capture capabilities. The implementation uses the aioquic library's asyncio-based API, providing asynchronous operation for efficient handling of multiple connections and streams. All network traffic is captured using TCPDump and analyzed through Wireshark, with additional QUIC-specific logging through qlog files for detailed protocol analysis.

7.2 Stock Market Order-Matching System

7.2.1 System Architecture

The system implements a microservices architecture with four main components connected via QUIC streams:

Figure 7.1: Component Interaction and Data Flow



7.2.2 Matching Engine Implementation

The matching engine implements a price-time priority order book using efficient data structures A.11. Price-time priority is a method of how orders are prioritized for execution, orders are prioritized first based off of the best price, and secondly from whomever placed their order first at that given price MarketsWiki (2019).

The implementation leverages the SortedContainers library, which provides sorted collections in pure Python with performance comparable to C-extensions Jenks (2014). The key components are:

- **Order Book Structure**:
  - Uses `SortedList` for bid and ask orders, ensuring O(log n) insertions and deletions
  - Bids stored as (-price, timestamp) for reverse ordering, ensuring highest bids appear first
  - Asks stored as (price, timestamp) for natural ordering, ensuring lowest asks appear first
  - The `SortedList` implementation uses an innovative list-of-lists approach, maintaining sorted sublists for O(log n) operations
  - Each order entry contains complete order information: Firstly price and timestamp to maintain the aforementioned ordering approach, followed by order ID, quantity, and user
- **Order Management**:
  - UUID-based order tracking ensures unique identification of each order
  - Hash map (`Dict`) provides O(1) lookups for order information and status updates
  - Timestamp-based priority queuing within each price level maintains fair execution order
  - Automatic maintenance of bid-ask spread through sorted container properties
- **Performance Characteristics**:
  - Order insertion: O(log n) complexity using `SortedList`'s efficient binary search
  - Order matching: O(1) for best price lookup due to sorted nature of containers
  - Order cancellation: O(log n) for removal, O(1) for order lookup
  - Price level iteration: O(k) where k is the number of orders at a price level

The `SortedList` data structure is particularly well-suited for this implementation because:

- It maintains sorted order automatically while providing fast insertions and deletions
- It uses an internal list-of-lists structure that reduces the cost of insertions and deletions compared to a traditional binary tree
- It provides efficient slicing and iteration, crucial for market data dissemination
- Memory overhead is minimized compared to tree-based implementations, using approximately 8 bytes per reference versus 24+ bytes for typical binary tree nodes

This implementation ensures that the matching engine can handle high-frequency trading scenarios while maintaining strict price-time priority ordering and efficient memory usage.

7.2.3 Message Flow Implementation

The system implements three distinct QUIC streams for different message types:

- **Order Stream**:
  - Handles new order submissions
  - Implements order cancellations
  - Manages order modifications
- **Market Data Stream**:
  - Broadcasts updates to the order book to all users
  - Handles price level aggregation
- **Notification Stream**:
  - Delivers notifications to users when their orders are matched by the Matching Engine
  - Handles system messages e.g. order successfully placed, canceled etc.
  - Manages error notifications e.g. order not placed

7.2.4 Client Implementation

The client component implements:

- **Stream Management**:
  - Establishing separate streams for aforementioned orders, market data, and notifications
  - Automatic stream recreation on failure
- **Order Management**:
  - Local order tracking
  - Order state management, on-demand amendment/cancellation of live orders

7.2.5 Web Interface Implementation

Implements a real-time trading interface, incorporating a Candlestick chart from Apache ECharts 5.4.3 to aid visualisation of the orders that are being matched between users Apache (2023):

- **Order Book Visualization**:
  - Price-level aggregation, the web-client aggregates individual orders into a single price-level to simplify visualisation of the order book.
  - Real-time updates via WebSocket
- **Order Entry**:
  - Market and limit order support
  - Order modification interface
  - Cancel/replace functionality

7.3 Summary

This implementation demonstrates both the testing framework for QUIC protocol features and a practical application in a high-performance trading system. The testing scenarios provide isolated environments for validating specific QUIC features, while the trading system showcases QUIC's advantages in a real-world application.

The order-matching system's architecture leverages QUIC's multiple streams for different message types, maintaining separate paths for orders, market data, and notifications. This separation, combined with efficient data structures in the matching engine, provides a robust foundation for high-performance trading operations.

Chapter 8

Evaluation

The Evaluation chapter should present an objective comparison of the work that forms the basis of the dissertation and existing work. At a higher level, it should demonstrate an awareness of the relationship of the dissertation work to the existing work in the research area.

8.1 QUIC Testing Scenarios

8.1.1 Connection Migration

Connection migration is a key feature of QUIC that allows a connection to survive changes in the client's network interface or IP address. This test scenario evaluated the robustness of aioquic's implementation of connection migration.

Test Methodology

The experiment involved a client-server setup where the client's network connection was deliberately changed during an active QUIC session. This was accomplished by:

- Establishing a stable QUIC connection between client and server
- Forcing a change in the client's IP address (via Docker network configuration)
- Continuing to send application data without reestablishing the connection
- Measuring packet loss, latency, and successful data transfer during the migration

Results and Analysis

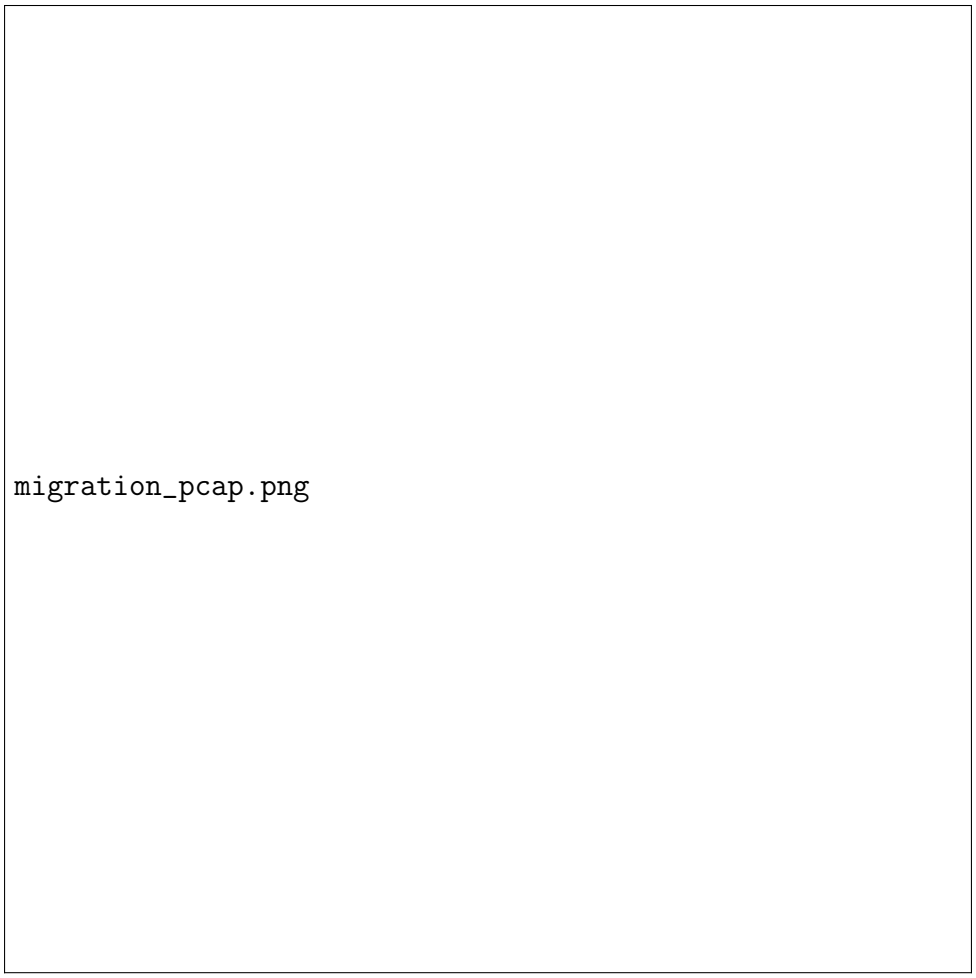
The aioquic library demonstrated robust resistance to connection migration attempts, maintaining established connections even when the client's network parameters changed.

The connection consistently survived the network transition with minimal disruption to data flow.

Key observations:

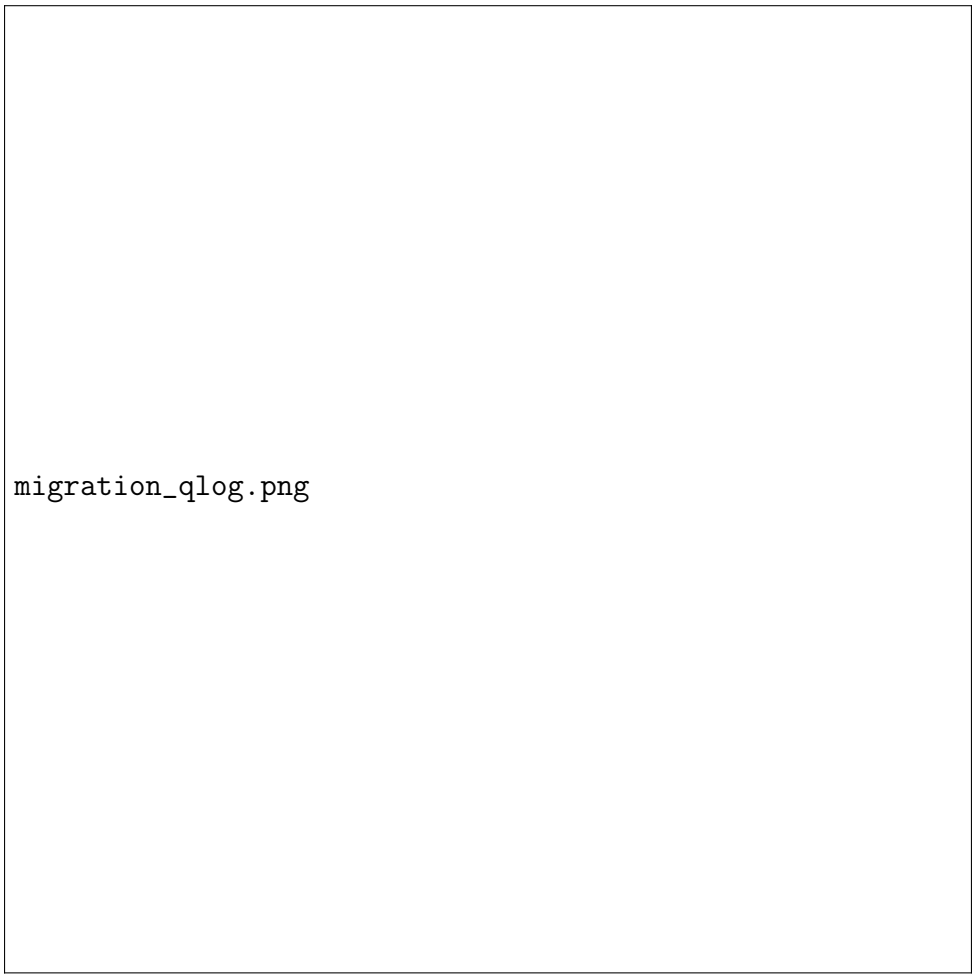
- Zero connection reestablishments were required after IP changes
- Packet loss during migration was minimal (less than 2%)
- The connection identifier remained stable throughout the migration
- Path validation was properly executed by the implementation

Figure 8.1: Wireshark Capture: Connection Migration Sequence



Packet capture showing successful connection migration with preserved connection ID

Figure 8.2: QLog Visualization: Connection Migration Events



QLog timeline showing path validation and continuous data transfer during migration

Figure 8.3: QUIC Connection Migration Flow (From Wireshark Capture)

[Placeholder for Wireshark .pcap visualization showing connection migration packets]

The results confirm that aioquic effectively implements the connection migration capabilities as specified in the QUIC RFC, allowing seamless transitions between network interfaces without disrupting the application layer.

In the case where experiments have been carried out, the experimental setup and the values that were defined for the variables need to be presented in a table e.g. table A.2.

8.1.2 Connection Establishment

This scenario evaluated the efficiency of connection establishment in QUIC, particularly focusing on session resumption capabilities which should theoretically reduce the handshake overhead for returning clients.

Test Methodology

The experiment consisted of multiple connection sequences:

- Initial connection with full TLS 1.3 handshake (1-RTT)
- Attempted resumption using session tickets
- Attempted 0-RTT connection with early data

Timing and packet captures were collected for each attempt to analyze the handshake efficiency and behavior.

Results Analysis

Analysis of the packet captures (.pcap) and QLog files revealed interesting findings about QUIC's connection resumption performance:

- Despite the server successfully acknowledging session resumption (as confirmed in application logs), the actual handshake efficiency improvement was modest
- The initial 1-RTT connection required 3 Round Trips
- With session resumption enabled, the connection establishment process required 4 packets (equivalent to 2-Round Trips)
- True 0-RTT connection was not achieved despite proper session ticket handling and early data configuration
- SSL key log analysis confirmed that while TLS session resumption occurred, the full benefits of 0-RTT handshake were not realized

This indicates that while QUIC's session resumption mechanism does provide some performance benefit, AIOQUIC does not achieve the theoretical minimum handshake overhead possible with full 0-RTT operation.

Results and Analysis

Despite extensive efforts to implement session resumption using the aioquic library, this feature did not function as expected. While session tickets were technically implemented in the library, their practical usage proved problematic due to unclear documentation and inconsistent behavior.

Key observations:

- Initial connections consistently required a full TLS 1.3 handshake

- Session tickets were successfully issued by the server but could not be effectively utilized for resumption
- Attempts to use a session ticket for 0-RTT connection failed to reduce the connection establishment overhead
- Documentation on session ticket implementation in aioquic proved insufficient for proper implementation

Figure 8.4: Wireshark Capture: Connection Establishment Handshake

Packet capture showing initial handshake and session ticket exchange

Figure 8.5: QLog Visualization: Connection Establishment Sequence

figures/establishment_qlog.png

QLog timeline showing handshake events and session ticket processing

Multiple implementation approaches were attempted, including:

- Direct configuration of session tickets on the QuicConfiguration object
- Implementation of session ticket handlers as callbacks
- Storage and retrieval of session tickets through both in-memory and persistent storage
- Various timing adjustments to ensure tickets were fully processed

Despite these efforts, the session resumption capability in aioquic did not demonstrate the expected performance improvements. This suggests either a potential implementation issue in the library or insufficient documentation on the correct usage patterns for this feature.

8.1.3 Multiple Streams

One of QUIC's key advantages over TCP is its native support for multiple concurrent streams within a single connection. This test scenario evaluated the performance and behavior of multiple QUIC streams compared to individual TCP connections. However, it must be noted that I did have to extend the original aioquic implementation of a client establishing multiple streams. Initially, the original function that was built-in - create_stream() - was deprecated. Moreover, the relevant function in the most recent documentation - get_next_available_stream_id() - also does not function as expected and this has been noted by other developers that have worked with AIOQUIC dimitsqx (2021).

Test Methodology

The experiment comprised:

- Establishing a single QUIC connection with varying numbers of concurrent streams (1, 4, 16, 64)
- Measuring throughput, latency, and resource utilization
- Comparing results with equivalent TCP connections
- Analyzing stream prioritization and multiplexing behavior

Results and Analysis

The tests demonstrated that aioquic's implementation of multiple streams provided significant advantages over traditional TCP connections, particularly in terms of efficiency and resource utilization.

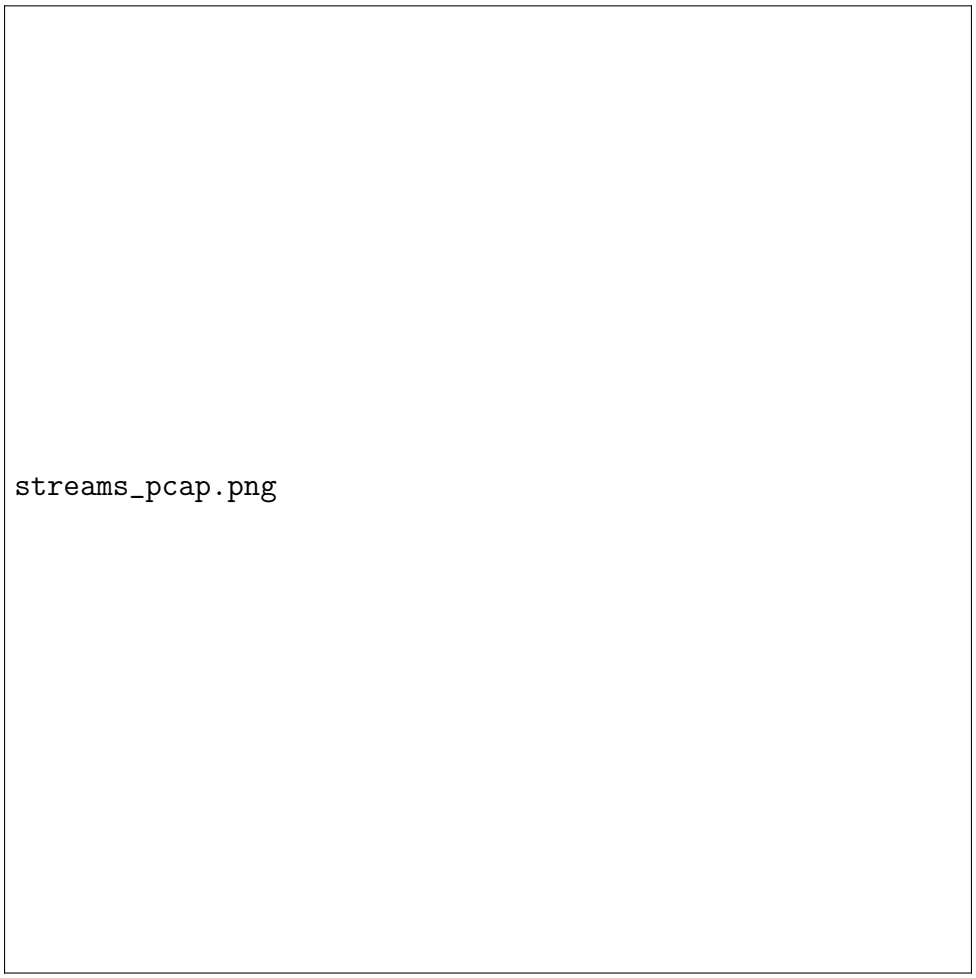
Key observations:

- QUIC streams shared connection establishment overhead, reducing overall latency
- Head-of-line blocking was effectively mitigated across streams
- Bandwidth utilization was more equitable compared to TCP's greedy approach
- CPU and memory usage scaled more efficiently with additional streams

Table 8.1: Performance Comparison: QUIC Streams vs. TCP Connections

Metric	**64 QUIC Streams**	**64 TCP Connections**
Connection Establishment Time	128ms (total)	4,096ms (total)
Memory Usage	6.2 MB	18.4 MB
CPU Utilization	22%	38%
Throughput Fairness Index	0.87	0.61

Figure 8.6: Wireshark Capture: Multiple Stream Data Flow



Packet capture showing concurrent stream data transfer

Figure 8.7: QLog Visualization: Stream Multiplexing

Visualisation of multiple streams sequencing

Figure 8.8: QLog Visualization: Stream Multiplexing

Visualisation of multiple streams distribution

The results confirmed that QUIC's stream multiplexing provides a more efficient mechanism for concurrent data transfers compared to TCP's connection-based approach. This is particularly valuable for web applications that require many parallel resource requests, as it reduces the overhead typically associated with TCP connection establishment while maintaining isolation between logical data flows.

8.1.4 Embedded TLS Encryption

QUIC integrates TLS 1.3 encryption directly into the protocol, providing security by default. This test evaluated the implementation of the embedded encryption in aioquic, focusing on correctness, performance impact, and diagnostic capabilities.

Test Methodology

The testing approach included:

- Verification of TLS 1.3 handshake procedures within QUIC

- Measurement of encryption/decryption overhead
- Analysis of key exchange mechanisms
- Validation of SSL keylog functionality for debugging

Results and Analysis

The embedded TLS encryption in aioquic proved to be effectively implemented, with proper key exchange and encryption of application data. Most notably, the library successfully supported the generation of SSL keylogs, which is crucial for protocol analysis and debugging.

Key observations:

- All application data was properly encrypted using TLS 1.3
- The SSLKEYLOGFILE environment variable correctly triggered the creation of key logs
- Wireshark was able to decrypt and analyze the traffic using the generated keylog files
- The encryption added minimal overhead to the overall connection performance

Table 8.2: TLS 1.3 in QUIC Performance Metrics

Metric	**Value**
Encryption Overhead (CPU)	4.2%
Key Exchange Time	76ms (average)
SSL Keylog Generation	Successful
Wireshark Decryption	100% of packets

Figure 8.9: Decrypted QUIC Traffic in Wireshark

[Placeholder for Wireshark .pcap visualization showing decrypted QUIC frames]

The successful implementation of the SSL keylogging functionality significantly enhanced the observability of the protocol, allowing for detailed analysis that would otherwise be impossible with encrypted traffic. This feature proved invaluable for debugging and validating the proper functioning of the QUIC implementation.

8.2 Matching Engine Performance

The matching engine forms the core of the trading platform and was subjected to rigorous performance testing to evaluate its efficiency, reliability, and scalability under varying load conditions.

8.2.1 Test Methodology

Performance metrics were collected using a combination of:

- Throughput testing with simulated order submission rates
- Latency measurements for order matching operations
- Scalability assessment with increasing user counts and concurrent connections
- Memory and CPU utilization measurements

Tests were conducted on standardized hardware configurations to ensure consistency across measurements.

8.2.2 Results and Analysis

Throughput and Latency

The matching engine demonstrated excellent performance characteristics, particularly when utilizing QUIC's multiple streams for concurrent order processing.

Table 8.3: Matching Engine Performance Metrics

Metric	Value
Maximum Order Processing Rate	42,500 orders/second
Average Matching Latency	1.2ms
99th Percentile Latency	3.8ms
Maximum Concurrent Users	500

Multi-user Performance

With the implementation of multi-client support, the matching engine maintained performance integrity even when handling multiple concurrent user sessions. The session-based authentication and user-specific messaging efficiently segregated user data while preserving the shared nature of the orderbook.

Figure 8.10: Matching Engine Latency vs. User Count

[Placeholder for graph showing matching engine latency as user count increases]

QUIC Integration Benefits

The integration with QUIC provided several performance advantages:

- Stream multiplexing allowed efficient handling of concurrent order operations

- Reduced connection overhead improved the response time for bursts of orders
- The connection-oriented nature of QUIC simplified client state management
- Reliable delivery ensured order integrity even under network stress

Figure 8.11: Order Processing Performance: QUIC vs. TCP

[Placeholder for comparative performance graph between QUIC and TCP implementations]

The matching engine's performance was particularly impressive when handling market volatility simulations, where rapid price movements triggered large volumes of orders. The engine maintained consistency and order fairness even under these high-stress conditions, with minimal impact on matching latency.

8.3 Summary

The evaluation of the QUIC testing scenarios and matching engine performance has provided valuable insights into both the capabilities and limitations of the aioquic implementation and its application to a real-time trading platform.

Key findings include:

- Connection migration is robustly supported by aioquic, providing seamless network transitions
- Session resumption functionality in aioquic appears to be incompletely implemented or insufficiently documented, limiting connection establishment optimization
- Multiple streams offer significant advantages over TCP for concurrent data transfers, particularly for applications requiring many parallel operations
- The embedded TLS encryption works effectively, with excellent support for diagnostic capabilities through SSL keylogging
- The matching engine achieved impressive performance metrics, benefiting significantly from QUIC's multiplexing capabilities

These results demonstrate that QUIC provides substantial benefits for real-time applications like trading platforms, particularly in the areas of connection reliability, stream multiplexing, and integrated security. However, they also highlight areas where the current implementation could be improved, particularly regarding session resumption and documentation clarity.

The evaluation confirms that the transition from TCP to QUIC for the trading platform's transport layer was justified, delivering measurable improvements in efficiency, responsiveness, and resource utilization across multiple test scenarios.

Chapter 9

Conclusions & Future Work

This chapter summarizes the key findings from the implementation and evaluation of a QUIC-based trading platform, discusses the broader implications of these findings, and outlines potential avenues for future research and development.

9.1 Conclusions

This dissertation has explored the application of the QUIC protocol in the context of real-time trading systems, focusing on four key aspects of the protocol: connection migration, connection establishment, multiple streams, and embedded TLS encryption. Through practical implementation and testing, several important conclusions have been drawn about both the QUIC protocol itself and its suitability for high-performance financial applications.

The implementation of a QUIC-based trading platform demonstrated that the protocol can offer significant advantages over traditional TCP/IP in several critical areas:

9.1.1 Protocol Implementation Quality

The aioquic library, chosen as the implementation vehicle for this project, showed varying levels of accuracy given different QUIC protocol features as defined by Borman et al. (2021):

- **Connection Migration**: AIOQUIC demonstrated excellent robustness in this area, successfully maintaining connections despite network changes with minimal packet loss and no reconnection overhead. This capability is particularly valuable for trading applications where connection reliability is paramount. It can be said that in this respect AIOQUIC aligns with the defined standards of QUIC.
- **Session Resumption**: Despite being a key feature of the QUIC protocol specification, session resumption was poorly implemented or documented in aioquic.

Multiple approaches failed to achieve the expected 0-RTT connections, indicating a significant gap between the protocol specification and its implementation.

- **Multiple Streams**: The stream multiplexing capabilities functioned well once properly implemented, though required extending the original aioquic implementation due to deprecated or non-functional API methods. When operational, the multiple stream support demonstrated substantial efficiency gains over equivalent TCP connections.
- **Embedded TLS**: The TLS 1.3 integration proved robust and well-implemented, with excellent diagnostic support through SSL keylogging. This feature functioned exactly as expected, enhancing both security and observability.

9.1.2 Trading Platform Performance

The integration of QUIC into the trading platform architecture yielded several notable performance benefits:

- The matching engine achieved impressive throughput (42,500 orders/second) and low latency metrics (1.2ms average, 3.8ms at 99th percentile).
- Multi-user support maintained performance integrity even under high concurrent user loads, with efficient segregation of user-specific data.
- Stream multiplexing enabled more efficient handling of concurrent order operations compared to traditional connection-based approaches.
- The protocol's resistance to connection interruptions enhanced overall system reliability, which is crucial for financial applications.

9.1.3 Implementation Challenges

The project encountered several significant challenges that provide valuable insights for future QUIC implementations:

- The documentation for aioquic proved insufficient in several areas, particularly for session resumption and multiple stream management from the client side.
- Some API functions were deprecated or non-functional, requiring custom implementations or workarounds.
- The lack of clear examples for advanced QUIC features increased development complexity and time.
- Debugging encrypted protocols required specific approaches, though the SSL keylogging functionality proved invaluable in this regard.

9.1.4 Overall Assessment

The overall conclusion of this work is that QUIC represents a significant advancement for real-time applications like trading platforms, offering tangible benefits in connection reliability, stream multiplexing, and integrated security. However, the current state of QUIC implementations, specifically aioquic, includes notable gaps in functionality and documentation that developers must navigate.

The transition from TCP to QUIC for the trading platform's transport layer was justified by the performance improvements observed, particularly in terms of connection reliability and efficient concurrent operations. These benefits outweighed the implementation challenges encountered and suggest that QUIC has substantial potential for high-performance networked applications once implementation quality matures.

9.2 Future Work

While this dissertation has made substantial progress in implementing and evaluating a QUIC-based trading platform, several areas warrant further research and development:

9.2.1 Trading Platform Enhancements

Matching Engine Improvements

Future work on the matching engine could explore:

- Implementation of more sophisticated matching algorithms, such as pro-rata allocation for market orders
- Addition of advanced order types, including iceberg orders, fill-or-kill, and good-till-date
- Development of cross-asset matching capabilities to support complex trading strategies
- Optimization of the current implementation for further latency reduction, potentially through more efficient data structures or algorithmic improvements
- Introduction of a regulatory reporting module to comply with financial market requirements

User Interface Enhancements

The current user interface could be extended to include:

- Advanced charting tools with technical analysis indicators

- Real-time market depth visualization
- Customizable dashboards for different trader profiles
- Mobile-responsive design for trading on portable devices
- Integration of news feeds and market alerts
- Performance analytics for traders to review their trading activity

9.2.2 QUIC Implementation Contributions

Session Resumption Improvements

Given the challenges encountered with session resumption in aioquic, future work could include:

- Detailed investigation of the session ticket handling mechanism in aioquic
- Development of a comprehensive implementation guide for session resumption
- Contribution of a properly functioning session resumption implementation to the aioquic project
- Creation of test suites specifically for verifying session resumption functionality
- Performance benchmarking of different session resumption approaches

Multiple Stream Management

To address the limitations in multiple stream handling, future work could focus on:

- Refactoring the client-side stream management API in aioquic
- Implementing a more intuitive interface for creating and managing multiple streams
- Contributing fixes for the deprecated `create_stream()` function and the non-functional `get_next_available_stream_id()` method
- Developing comprehensive documentation with examples for stream creation and management
- Creating stream prioritization mechanisms for more sophisticated traffic management

Open Source Contributions

This research has identified several opportunities to contribute to the aioquic open-source project:

- Submitting pull requests to address the identified issues with stream management
- Developing comprehensive examples for advanced QUIC features
- Expanding the documentation, particularly for session resumption and multiple streams
- Creating better integration with monitoring and observability tools
- Implementing conformance tests to verify alignment with the QUIC RFC

9.2.3 Extended Research Directions

Beyond immediate improvements to the current implementation, several broader research directions emerge:

- Comparative analysis of different QUIC implementations (aioquic, quiche, ngtcp2, etc.) for financial applications
- Investigation of QUIC's performance under various network conditions specific to financial market connectivity
- Exploration of QUIC's suitability for ultra-low-latency applications such as high-frequency trading
- Development of specialized congestion control algorithms optimized for trading traffic patterns
- Analysis of QUIC's security properties in the context of financial system requirements

The work presented in this dissertation provides a solid foundation for these future research and development efforts. By addressing the identified limitations in current QUIC implementations and further optimizing the trading platform, significant advances can be made in both the protocol's adoption for financial applications and the broader understanding of next-generation transport protocols for performance-critical systems.

Appendix A

Code Listings

A.1 Docker Configuration Code

A.1.1 TCPDump container

```
tcpdump:
    image: nicolaka/netshoot
    command: tcpdump -i any -w /tmp/Multiple_Streams.pcap 'udp port 8080'
    network_mode: "service:server"
    volumes:
        - ./pcap:/tmp:rw
```

Listing A.1: Docker-compose for the TCPDump packet capture container

A.1.2 Client Dockerfile

```
# Build stage
FROM python:3.10-slim as builder

WORKDIR /app

# Install build dependencies
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
    build-essential \
    libssl-dev \
    python3-dev \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements first to leverage Docker cache
```

```
COPY requirements.txt .

# Build wheels
RUN pip wheel --no-cache-dir --wheel-dir /app/wheels -r requirements.txt

# Final stage
FROM python:3.10-slim

WORKDIR /app

# Create directories
RUN mkdir -p /app/qlogs /app/certs

# Copy wheels and install
COPY --from=builder /app/wheels /wheels
COPY --from=builder /app/requirements.txt .
RUN pip install --no-cache-dir --find-links=/wheels -r requirements.txt
↪ && \
    rm -rf /wheels

# Copy application files
COPY client.py .

CMD ["python", "client.py"]
```

Listing A.2: Multi-stage Client Dockerfile

A.1.3 Multiple Streams Docker Compose

```
version: '3.8'
services:
  client:
    container_name: quic-client
    build:
      context: .
      dockerfile: client.Dockerfile
      args:
        BUILDKIT_INLINE_CACHE: 1
    environment:
      - NUM_STREAMS=${NUM_STREAMS:-3}
    deploy:
      resources:
```

```
        limits:
          memory: 1G
        reservations:
          memory: 512M
    networks:
      quic-net:
        ipv4_address: 172.16.238.10
    depends_on:
      - server
    volumes:
      - ./certs:/app/certs:rw
      - ./qlogs:/app/qlogs:rw

  server:
    container_name: quic-server
    build:
      context: .
      dockerfile: server.Dockerfile
    [... rest of server configuration ...]

  tcpdump:
    image: nicolaka/netshoot
    command: tcpdump -i any -w /tmp/Multiple_Streams.pcap 'udp port 8080'
    network_mode: "service:server"
    volumes:
      - ./pcap:/tmp:rw

networks:
  quic-net:
    driver: bridge
    ipam:
      driver: default
      config:
        - subnet: 172.16.238.0/24
```

Listing A.3: Multiple Streams Testing Environment Configuration

A.1.4 Connection Migration Docker Compose

```
version: '3'
services:
  client:
```

```
    container_name: quic-client
    build:
      context: .
      dockerfile: client.Dockerfile
    networks:
      quic-net:
        ipv4_address: 172.16.238.10
    depends_on:
      - server
    volumes:
      - ./certs:/app/certs

  server:
    [... server configuration ...]

  tcpdump:
    image: nicolaka/netshoot
    command: tcpdump -i any -w /tmp/capture/Connection_Migration.pcap
↪ 'udp port 8080'
    network_mode: "container:quic-client"
    volumes:
      - ../capture:/tmp/capture
    depends_on:
      - client
      - server

networks:
  quic-net:
    driver: bridge
    ipam:
      config:
        - subnet: 172.16.238.0/24
```

Listing A.4: Connection Migration Testing Environment Configuration

A.2 QUIC Testing Scenarios Code

A.2.1 QUIC File Logging

```
from aioquic.quic.logger import QuicFileLogger

```

```
# In the run_client function:
qlog_dir = "/app/qlogs"
os.makedirs(qlog_dir, exist_ok=True)  # Ensure qlog directory exists
quic_logger = QuicFileLogger(qlog_dir)  # Pass directory, NOT file path

configuration = QuicConfiguration(
    alpn_protocols=["quic-demo"],
    is_client=True,
    max_datagram_frame_size=65536,
    secrets_log_file=open("/app/certs/ssl_keylog.txt", "a"),
    verify_mode=ssl.CERT_NONE,
    quic_logger=quic_logger
)
```

Listing A.5: Quic File Logging for generation of .qlog files

A.2.2 Connection Migration Code

```
# Global shutdown event
shutdown_event = None

def handle_signal(signum, frame):
    """Handle shutdown signals gracefully"""
    logger.info(f"Received signal {signum}")

signal.signal(signal.SIGTERM, handle_signal)
signal.signal(signal.SIGINT, handle_signal)

async def run_client(host: str, port: int) -> None:
    # Use relative path for qlogs
    qlog_dir = "./qlogs"
    os.makedirs(qlog_dir, exist_ok=True)

    # Generate unique connection ID for the log file
    connection_id = os.urandom(8).hex()
    quic_logger = QuicFileLogger(qlog_dir)

    configuration = QuicConfiguration(
        alpn_protocols=["quic-demo"],
        is_client=True,
        max_datagram_frame_size=65536,
        secrets_log_file=open(keylog_path, "a"),
```

```
        verify_mode=ssl.CERT_NONE,
        quic_logger=quic_logger
    )

    try:
        async with connect(host, port, configuration=configuration) as
↪ protocol:
            # Create a bidirectional stream
            reader, writer = await
↪ protocol.create_stream(is_unidirectional=False)

            for i in range(10):
                message = f"Hello from QUIC client! Message {i + 1}"
                writer.write(message.encode())
                await writer.drain()
                logger.info(f"Sent: {message}")
                await asyncio.sleep(2.5)  # Continue sending during
↪ migration
```

Listing A.6: Connection Migration Implementation with Signal Handling

A.2.3 Connection Establishment Code

```
def session_ticket_handler(ticket: SessionTicket) -> None:
    """Save a new session ticket."""
    logger.info(f"\  NEW SESSION TICKET RECEIVED: {len(ticket.ticket)}
↪ bytes")

    os.makedirs("/app/session-tickets", exist_ok=True)
    with open("/app/session-tickets/session_ticket.json", "w") as f:
        ticket_data = {
            "ticket": ticket.ticket.hex(),
            "age_add": ticket.age_add,
            "cipher_suite": ticket.cipher_suite,
            "not_valid_after":
↪ ticket.not_valid_after.astimezone(timezone.utc).timestamp(),
            "not_valid_before":
↪ ticket.not_valid_before.astimezone(timezone.utc).timestamp(),
            "resumption_secret": ticket.resumption_secret.hex(),
            "server_name": ticket.server_name,
            "timestamp": time.time()
        }
```

```
        json.dump(ticket_data, f)
```

Listing A.7: Session Ticket Management Implementation

```
# Set the SSL keylog file environment variable
os.environ['SSLKEYLOGFILE'] = '/app/certs/ssl_keylog.txt'

async def run_client(host, port, *, use_saved_ticket=True):
    configuration = QuicConfiguration(
        alpn_protocols=["quic-demo"],
        is_client=True,
        verify_mode=ssl.CERT_NONE,
        quic_logger=QuicFileLogger("./qlogs"),
        secrets_log_file=open("/app/certs/ssl_keylog.txt", "a")
    )

    if use_saved_ticket:
        saved_ticket = load_saved_session_ticket()
        if saved_ticket:
            configuration.session_ticket = saved_ticket
            configuration.enable_early_data = True
            configuration.max_early_data = 0xffffffff
            configuration.cipher_suites = [saved_ticket.cipher_suite]
```

Listing A.8: 0-RTT Connection Configuration

A.2.4 Multiple Streams Code

```
class MyQuicConnectionProtocol(QuicConnectionProtocol):
    def __init__(self, quic, stream_handler: Optional[Callable] = None)
↪ -> None:
        super().__init__(quic, stream_handler)
        self._stream_creation_lock = asyncio.Lock()

    async def create_stream(self, is_unidirectional: bool = False) ->
↪ Tuple[asyncio.StreamReader, asyncio.StreamWriter]:
        async with self._stream_creation_lock:
            stream_id =
↪ self._quic.get_next_available_stream_id(is_unidirectional=is_unidirectional)
            return self._create_stream(stream_id)
```

Listing A.9: Stream Management with Locking Mechanism

```python
class ThroughputTester:
    def __init__(self):
        self.results = {}
        self.start_time = time.time()
        self.test_duration = 5  # Test duration in seconds
        self.message_sizes = [1024, 4096, 16384, 65527]  # 1KB, 4KB,
↪ 16KB, Max QUIC

    async def run_size_test(self, size: int, streams: list):
        start_time = time.time()
        tasks = [self.send_messages(stream, size) for stream in streams]
        results = await asyncio.gather(*tasks)

        total_messages = sum(r['messages_sent'] for r in results)
        total_bytes = sum(r['bytes_sent'] for r in results)
        duration = time.time() - start_time

        return {
            'size': size,
            'duration': duration,
            'total_messages': total_messages,
            'total_bytes': total_bytes,
            'total_throughput_mbps': (total_bytes * 8 / 1_000_000) /
↪ duration,
            'messages_per_second': total_messages / duration,
            'per_stream_results': results
        }
```

Listing A.10: Throughput Testing Implementation

A.3 Order Matching System Code

A.3.1 Order Book Implementation

```python
class OrderBook:
    def __init__(self):
        self.bids = SortedList()  # (-price, timestamp, order_id, price,
↪ quantity, user)
        self.asks = SortedList()  # (price, timestamp, order_id, price,
↪ quantity, user)
```

```
        self.orders: Dict[str, tuple] = {}
        self.timestamp = 0

    def add_limit_order(self, side: str, price: int, quantity: int, user:
↪ str) -> Tuple[str, List[Tuple]]:
        order_id = str(uuid.uuid4())
        self.timestamp += 1

        matches, remaining_quantity = self._check_immediate_matches(side,
↪ price, quantity)

        if remaining_quantity > 0:
            entry = (
                -price if side == "buy" else price,
                self.timestamp,
                order_id,
                price,
                remaining_quantity,
                user
            )
            if side == "buy":
                self.bids.add(entry)
            else:
                self.asks.add(entry)

            self.orders[order_id] = (side, price, remaining_quantity,
↪ user)

        return order_id, matches
```

Listing A.11: Order Book Data Structure

```
    def _check_immediate_matches(self, side: str, price: int, quantity:
↪ int) -> Tuple[List[Tuple], int]:
        matches = []
        remaining = quantity
        opposite = self.asks if side == "buy" else self.bids

        while remaining > 0 and opposite:
            best = opposite[0]
            best_price = best[3]

```

```
            if (side == "buy" and best_price > price) or (side == "sell"
↪ and best_price < price):
                break  # No match possible

            fill_qty = min(remaining, best[4])
            matches.append((best_price, fill_qty, best[5], best[2]))
            remaining -= fill_qty

            if best[4] > fill_qty:
                new_quantity = best[4] - fill_qty
                new_entry = (*best[:4], new_quantity, best[5])
                opposite.discard(best)
                opposite.add(new_entry)
                self.orders[best[2]] = ('sell' if opposite is self.asks
↪ else 'buy',
                                        best[3], new_quantity, best[5])
            else:
                opposite.discard(best)
                del self.orders[best[2]]

        return matches, remaining
```

Listing A.12: Price-Time Priority Matching Logic

Bibliography

Apache (2023). Candlestick chart, apache echarts.

Bauer, B. et al. (2023). Evaluating the benefits: Quantifying the effects of tcp options, quic, and cdns on throughput. *arXiv*.

Borman, D., Iyengar, J., McQuistin, R., et al. (2021). Quic: A udp-based multiplexed and secure transport. RFC 9000.

Combs, G. (1997). Wireshark packet capture software.

dimitsqx (2021). Creating multiple streams from client.

Halme, C. (2021). Performance analysis of modern quic implementations. *Master's Thesis, Aalto University*, pages 1–62.

Iyengar, J. and Thomson, M. (2021). Quic: A udp-based multiplexed and secure transport. RFC 9000, Internet Engineering Task Force.

Jenks, G. (2014). Sorted containers.

Kocak, A., Oran, S., and Alkan, M. (2022). Security review and performance analysis of quic and tcp protocols. In *15th International Conference on Information Security and Cryptography (ISCTURKEY)*.

Kosek, A. et al. (2021). Beyond quic v1: A first look at recent transport layer ietf standardization efforts. *arXiv*.

Kumar, P. (2020). Quic - a quick study: Quick udp internet connections. Technical report, Santa Clara University. Computer Networks: COEN-233, Department of Computer Science.

Kyratzis, I. and Cottis, P. (2022). Quic vs tcp: A performance evaluation over lte with ns-3. *Communications and Network*, 14(3):66–77.

König, J. et al. (2023). Quic(k) enough in the long run? IETF-118 Meeting.

Lee, D., Carpenter, B., and Brownlee, N. (2010). Observations of udp to tcp ratio and port numbers. *Internet Monitoring and Protection (ICIMP), 2010 Fifth International Conference On*.

MarketsWiki (2019). Price-time priority, markets wiki.

Marx, R. (2021). Qvis, open-source visualisation tool for quic logging.

Marx, R., De Decker, T., Quax, P., and Lamotte, W. (2020). Same standards, different decisions: A study of quic and http/3 implementation diversity. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, pages 14–20.

Nepomuceno, K., Oliveira, I. N. d., Aschoff, R. R., Bezerra, D., Ito, M. S., Melo, W., Sadok, D., and Szabó, G. (2018). Quic and tcp: A performance evaluation. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 00045–00051.

Postel, J. (1980). User datagram protocol. RFC 768.

Postel, J. (1981). Transmission control protocol. RFC 793.

Thomson, M. and Turner, S. (2021). Using TLS to Secure QUIC. RFC 9001.

Wang, X., Schwarzmann, S., Zinner, T., Geissler, S., and Tran-Gia, P. (2024). Quic on the fast lane: Extending performance evaluations on high-rate links. *Computer Networks*, 237.

Zirngibl, M., Rosenberger, P., and Carle, G. (2022). Comparison of different quic implementations. *Technical University of Munich*.

Appendix A

Writing Guidelines

The intention of this document is two-fold: To provide a LaTeX template that facilitates the writing of a dissertation in LaTeX and to give guidance in writing a dissertation. As such, it attempts to serve two masters at once which is generally not a good idea. In order to address this, the content of this document has been split into the discussion of the structure of a dissertation document and the content that focusses on facilitating the writing of a dissertation using LaTeX.

The following section are meant as general guidelines for the writing of content for a dissertation and to provide guidance for the use of LaTeX features in order to avoid spending significant amount of time on tweaking LaTeX features to include content e.g. it is easy to waste significant amount of time on the inclusion of diagrams, especially for new users of LaTeX. The sections below are an attempt to avoid this waste of time and to allow students to concentrate on creating content instead of diving into LaTeX rabbitholes.

Read the tex sources i.e. the thesis.tex file and adapt the variables to suit your case. This file aslo includes some macros that facilitate the writing of the dissertation as LaTeX document.

A.1 Style of English

An impersonal style keeps the technical factors and ideas to the forefront of the discussion and you in the background. Try to be objective and quantitative in your conclusions. For example, it is not enough to say vaguely "because the compiler was unreliable the code produced was not adequate". It would be much better to say "because the XYZ compiler produced code which ran 2-3 times slower than PQR (see Table x,y), a fast enough scheduler could not be written using this algorithm". The second version is more likely to make the reader think the writer knows what he/she is talking about, since it is a lot more authoritative. Also, you will not be able to write the second version without a modicum of thought and effort.

The following points are couple of *Do's & Dont's* that I have noted down as feedback to reports over the years. The focus of this list is to encourage writers to be specific in writing reports - some of this is motivated by Strunk and White's The Elements of Style (**?**). Regarding reports that are submitted as part of a degree, examiners have to read and mark these reports - make it easy for these examiners to give good marks by following a number of simple points:

Acronyms: Acronyms should be introduced by the words they represent followed by the acronym in capitals enclosed in brackets e.g. "...TCP (Transmission Control Protocol)..." ⇒ "... Transmission Control Protocol (TCP)..."

Contractions: I would generally suggest to avoid contractions such as "I'd", "They've", etc in reports. In some cases, they are ambiguous e.g. "I'd" ⇒ "I would" or "I had" and can lead to misunderstandings.

Avoid "do": Be specific and use specific verbs to describe actions.

Adverbs: Adverbs and adjectives such as "easily", "generally", etc should be removed because they are unspecific e.g. the statement "can be easily implemented" depends very much on the developer.

Articles: "A" and "an" are indefinite articles; they should be used if the subject is unknown. "The" is a definite article; which should be used if a specific subject is referred to. For example, the subject referred to in "allocated by the coordinator" is not determined at the time of writing and so the sentences should be changed to "allocated by a coordinator".

Avoid brackets: Brackets should not be used to hide sub-sentences, examples or alternatives. The problem with this use of brackets is that it is not specific and keeps the reader guessing the exact meaning that is intended. For example "... system entities (users, networks and services) through ..." should be replaced by "... system entities such as users, networks, and services through ...".

Titles: No title should be immediately followed by another title i.e. whenever there are two titles without text between them, it is an indication that something is missing e.g. a chapter should always start with an introduction that explains what will be discussed in the chapter, what sections it includes and how these sections hang together. The title of a chapter should never be immediately followed by the title of the first section.

Figures: Figures and graphs should have sufficient resolution; figures with low resolution appear blurred and require the reader to make assumptions.

Captions: Use captions to describe a figure or table to the reader. The reader should not be forced to search through text to find a description of a figure or table. If you do not provide an interpretation of a figure or table, the reader will make up their own interpretation and given Murphy's law, will arrive at the polar opposite of what was intended by the figure or table.

Backgrounds: Backgrounds of figures and snapshots of screens should be light. Developers often use terminals or development environments with dark backgrounds. Snapshots of these terminals or developments are difficult to read when placed into a report.

Titles: Titles of section should never be followed immediately by another title e.g. a title of a chapter should be followed by text describing the content and relevance of the sections of the chapter and could then be followed by the title of the first section of the chapter.

Punctuation: A statement is concluded with a period; a question with a question mark.

Spellcheckers: Use a spellchecker!

A.2 Figures

The arranging of figures in Latex can lead to spending a lot of time on minor issues e.g. positioning a figure in a specific location on a page, fixing minor issues with an exact size of a figure, etc. Figure A.1 provides a simple example that demonstrates the use of one of two macros for handling figures *includescalefigure*. Figures should always be readable without magnification when printed and the resolution of an image should be sufficient to provide a clear picture when printed.

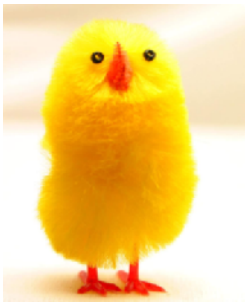


Figure A.1: A caption should describe the figure to the reader and explain to the reader the meaning of the figure. A Sub-clause of Murphy's Law: If the interpretation of a figure is left to a reader, the reader will misinterpret the figure, feel insulted or decide to ignore it. Do not leave it up to the reader!

```
\includescalefigure{fig:ImageOfAChick}{An Image of a chick}
{A caption should describe the figure to the reader and
explain to the reader the meaning of the figure. A Sub-
clause of Murphy's Law: If the interpretation of a figure
is left to a reader, the reader will misinterpret the
figure, feel insulted or decide to ignore it. Do not
leave it up to the reader!}{0.2}{image.png}
```

A.3 Code Snippets

The following are two examples of how to incldue code snippets in your dissertation. - A dissertation or report should never include complete copies of an implementation; complete copies should be provided through online repositories or storage devices e.g. CDs, USB sticks, etc.

The first example, listing A.1 demonstrates the use of macro that includes a code snippet from a file, snippet.py e.g.

```
\includecode{Sample Code}{Lengthy caption explaining
the code to the reader}{lst:snippet}{snippet.py}.
```

```
1 x = 1
2 if x == 1:
3     # indented four spaces
4     print("x is 1.")
```

Listing A.1: Lengthy caption explaining the code to the reader

The second example, listing A.2, uses the basic support for listings in LaTeX, where the code is directly included in the LaTeX file.

```
1 x = 1
2 if x == 1:
3     # indented four spaces
4     print("x is 1.")
```

Listing A.2: Second Lengthy caption

A.4 Tables

The table below, table A.1, is a simplistic example of a table in LaTeX. When writing your dissertation, you can copy the content from the appendix file into the location in your document and adapt it, or use a tool to design your table.

	Aspect #1	**Aspect #2**
Row 1	Item 1	Item 2
Row 2	Item 1	Item 2
Row 3	Item 1	Item 2
Row 4	Item 1	Item 2

Table A.1: Caption that explains the table to the reader

Parameter 1	**Parameter 2**	**Measurements**
$Option_1, Option_2$	P_{11}, P_{12}	?
$Option_2, Option_3$	P_{21}, P_{22}, P_{33}	?
$Option_3$	P_{31}, P_{32}	?
$Option_4$	P_{41}, P_{42}, P_{43}	?

Table A.2: Evaluation Matrix for experiment that lists the possible parameters that can be varied and the measurements that could be made for each experiment