BTP Project Proposal: A Comparative Study of QUIC at the Kernel Level

Swapnil Garg 22115150

Mmukul Khedekar 22114054

September 23, 2025

Abstract

This document outlines a project proposal for a comparative study of the QUIC transport protocol. The project aims to compare the performance of application-level QUIC, which is the current standard, with a custom-built kernel-level QUIC implementation and, to provide a detailed analysis of the associated advantages and disadvantages of this approach. The performance of both QUIC implementations will be benchmarked against the traditional TCP transport protocol. The study will focus on key performance metrics such as throughput, latency, CPU utilization, and packet loss recovery. The ultimate goal is to determine if a kernel-level implementation of QUIC can offer significant performance advantages over its user-space counterpart such as by reducing context switching overhead and as well as investigate other areas of optimisations and improvements. This proposal also details the project timeline, methodology, and a review of existing literature.

Contents

1 Introduction **3**

2 Literature Review **4**
2.1 Relevant Standards and Discussions 4
2.2 Key Findings in Existing Literature 5
2.3 Research Gap 5

3 Project Proposal and Methodology **6**
3.1 Phase 1: Application-level QUIC and TCP Baseline 6
3.2 Phase 2: Kernel-level Implementation and Learning 6
3.3 Phase 3: Comparative Analysis and Final Report 6
3.3.1 Key Performance Metrics 6

4 Timeline and Deliverables **7**

5 Conclusion and Future Work **8**

1 Introduction

The rise of modern web applications and real-time services has created a demand for transport protocols that are more efficient and secure than the traditional TCP/IP stack. The QUIC (Quick UDP Internet Connections) protocol, originally developed by Google, has emerged as a promising solution. Built on top of UDP, QUIC provides several key improvements, including reduced connection establishment latency, stream multiplexing to prevent head-of-line blocking, and integrated TLS 1.3 encryption. QUIC was designed to address the shortcomings of TCP, with the long-term goal of making TCP obsolete.

Over the years, QUIC has gained rapid and widespread attention, having been standardized as the transport layer protocol for HTTP/3 by the IETF in 2021. Significant efforts are underway to optimize QUIC, and it is being adopted rapidly as a transport protocol.

Despite its growing adoption, almost all production-ready implementations of QUIC remain in user space, with no support in the kernel. User-space deployment enables rapid development and deployment cycles, as updates do not require modifications to the operating system kernel. However, this approach introduces performance overhead due to repeated context switches and data copying between user and kernel spaces.

This project proposes to explore the potential benefits of a kernel-level QUIC implementation. By moving the protocol logic directly into the kernel, we hypothesize that it will be possible to reduce the processing overhead and achieve a more streamlined data path, leading to superior performance in comparison to both TCP and application-level QUIC.

2 Literature Review

In this section, we present an overview of relevant work that has been done in this research area. We will highlight the research gaps and reiterate the aim of the project.

2.1 Relevant Standards and Discussions

Topic / Source	Key Contributions & Findings	Links
QUIC RFCs (RFC 9000, 9001, 9002)	Defines the core protocol as a UDP-based, multiplexed, and secure transport. Key features include stream multiplexing to avoid head-of-line blocking, integrated TLS 1.3, and low-latency connection establishment (0-RTT).	`https://datatracker.ietf.org/doc/rfc9000/` `https://datatracker.ietf.org/doc/html/rfc9001` `https://datatracker.ietf.org/doc/html/rfc9002`
Motivation for Kernel QUIC	Discusses performance benefits of reducing user-kernel context switches. Highlights the goal of making QUIC available to a wider range of applications without requiring application-level libraries.	`https://news.ycombinator.com/item?id=29564158` `https://lwn.net/Articles/1029851/`
Socket API Proposal	Proposes standard socket APIs for QUIC (`IPPROTO_QUIC`), covering operations like `listen`, `connect`, and `sendmsg`. Aims for a familiar programming model similar to TCP.	`https://www.ietf.org/archive/id/draft-lxin-quic-socket-apis-00.html`
Kernel QUIC Implementation Proposal	Describes a hybrid architecture: core QUIC protocol in the kernel, but the complex TLS handshake is processed in user space via a helper daemon.	`https://lwn.net/Articles/989623/` `https://github.com/lxin/quic`

2.2 Key Findings in Existing Literature

- **Performance in Lossy Networks:** Multiple studies demonstrate that QUIC achieves superior throughput and faster connection establishment in high-latency and lossy environments compared to TCP.
- **Head-of-Line Blocking:** QUIC's native support for stream multiplexing has been shown to effectively mitigate Head-of-Line (HoL) blocking, one of TCP's major limitations arising from its single, ordered byte stream model.
- **CPU Overhead:** A recurring observation across studies is the relatively higher CPU utilization of user-space QUIC implementations compared to in-kernel TCP. This is largely due to the lack of kernel-level optimizations (e.g., TCP Segmentation Offload) for UDP traffic, as well as the computational overhead of handling protocol logic in user space.
- **Kernel-Level QUIC:** Kernel-level implementations of QUIC are currently limited to experimental proposals. This remains a highly active area of research, focusing on performance analysis, investigation of trade-offs, and exploration of potential optimizations. Several feasibility studies are also underway.

2.3 Research Gap

While QUIC has been extensively studied and standardized at the application level, production-ready implementations are still primarily limited to user space. Kernel-level implementations remain experimental, with limited benchmarking and performance evaluation available. Our project aims to extend existing research by analyzing performance trade-offs and exploring possible advantages and optimizations of implementing QUIC in the Kernel.

3 Project Proposal and Methodology

Our project will involve three main phases to accomplish the comparative study.

3.1 Phase 1: Application-level QUIC and TCP Baseline

The first phase will focus on implementing a simple client-server model of QUIC. This implementation will serve as one of the baselines for our comparison. We will use this setup to collect initial performance data for both application-level QUIC and a standard TCP connection under controlled network conditions. The primary goal of this phase is to gain proficiency in the QUIC protocol specification and its advantages over TCP, while also implementing it practically and conducting a comparative study in userspace.

3.2 Phase 2: Kernel-level Implementation and Learning

During the winter break, we will dedicate time to understanding and learning Linux kernel programming. This is the most crucial part of the project. We will focus on the following:

- Understanding the Linux network stack architecture.
- Writing and testing basic kernel modules.
- Designing the architecture of our kernel-level QUIC implementation to handle core protocol features such as packet parsing, stream management, and congestion control.

3.3 Phase 3: Comparative Analysis and Final Report

In the final phase, we will implement the designed kernel-level QUIC stack. Once the implementation is complete and functional, we will conduct a series of controlled experiments to compare the three protocols: kernel-level QUIC, application-level QUIC, and TCP.

3.3.1 Key Performance Metrics

The comparison will be quantitative and based on the following metrics:

- **Throughput:** Measured as the data transfer rate in megabits per second (Mbps). We will conduct tests under varying bandwidth limits.
- **Latency:** Measured as Round-Trip Time (RTT) for data packets and total time for connection establishment. We will introduce artificial network delays to observe performance under different conditions.
- **CPU Utilization:** Measured by monitoring CPU usage of the server process to quantify the overhead of each protocol stack.
- **Packet Loss Recovery:** Measured by introducing controlled packet loss and observing the time taken to recover and the impact on overall throughput.
- **Jitter:** Measured as the variation in packet delay to understand the suitability of each protocol for real-time applications.

4 Timeline and Deliverables

Phase	Tasks & Deliverables	Timeline
Phase 1 (Current Semester)	**Application-level Implementation** • Study the QUIC protocol specification • Set up a local test environment • Implement client-server application that uses QUIC • Conduct initial performance tests for baseline data • **Deliverable:** Working user-space QUIC + performance data	**Ongoing**
Phase 2 (Winter Break)	**Kernel Programming and Design** • Learn Linux kernel programming fundamentals • Study Linux network stack and kernel module development • Design architecture for kernel-level QUIC implementation • **Deliverable:** Detailed design document for kernel-level stack	**Nov-Dec**
Phase 3 **(Next Semester)**	**Kernel-level Implementation and Evaluation** • Implement kernel-level QUIC stack as a module • Conduct comprehensive performance tests on all protocols • Analyze collected data and interpret results • Write final project report with findings • Prepare final presentation • **Deliverable:** Working kernel-level QUIC prototype + final report	**Jan-Apr**

5 Conclusion and Future Work

This project aims to provide a definitive comparison between the performance characteristics of QUIC implemented in user space versus the kernel, using TCP as a well-established benchmark. The findings from this study will contribute to the ongoing discussion about the optimal deployment for modern transport protocols. Future work could include implementing more advanced QUIC features like multipath support or exploring hardware offloading capabilities for a kernel-level implementation.