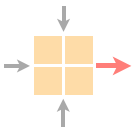
ETH zürich



Networked Systems
ETH Zürich — seit 2015

# QUIC network stack optimization using syscall batching

Semester Thesis

Author: Hao Zhu

Tutor: Laurin Brandner

Supervisor: Prof. Dr. Laurent Vanbever

October 2024 to January 2025

**Abstract**

QUIC is a next-generation transport protocol designed to address the limitations of traditional transport protocols. However, its processing overhead becomes its main performance bottleneck. In this work, we leverage `io_uring`, an asynchronous interface to the Linux kernel, to batch system calls and reduce context switches made by the QUIC server. This approach reduces processing overhead and enhances server performance. Experimental results demonstrate that, under an ideal high-bandwidth and lossless network, the QUIC server with the `io_uring` network stack achieves a 24.4% reduction in download time for an 80 MB object, maintains 27.8% higher bandwidth, and invokes significantly fewer system calls. Moreover, the number of system calls grows at a much slower rate as the number of connections increases, indicating improved scalability.

# Contents


**1 Introduction** **1**
- 1.1 Motivation . . . 1
- 1.2 Task and Goals . . . 1
- 1.3 Overview . . . 1

**2 Background and Related Work** **2**
- 2.1 Background . . . 2
  - 2.1.1 QUIC . . . 2
  - 2.1.2 io_uring . . . 2
- 2.2 Related Work . . . 3

**3 Design** **4**
- 3.1 QUIC io_uring Server Design . . . 4
  - 3.1.1 Server Architecture . . . 4
  - 3.1.2 Receiver thread . . . 5
  - 3.1.3 Worker thread . . . 5
- 3.2 Baseline Server Design . . . 7
- 3.3 Client Design . . . 7

**4 Evaluation** **8**
- 4.1 Experiment setup . . . 8
- 4.2 Result . . . 9
  - 4.2.1 Time to Last Byte (TTLB) . . . 9
  - 4.2.2 Bandwidth . . . 10
  - 4.2.3 System Calls . . . 11

**5 Outlook** **13**

**References** **14**

# Chapter 1

# Introduction

## 1.1 Motivation

QUIC is a next-generation transport protocol designed to address the limitations of traditional transport protocols, aiming to reduce latency and enhance efficiency. Currently, over 8% of global websites utilize QUIC [14]. To circumvent middlebox interference and enable rapid deployment of updates without OS modifications, QUIC is implemented in user-space and operates atop UDP [9]. The user-space design offers flexibility but also presents certain challenges. QUIC servers experience approximately 3.5 times higher utilization than TLS/TCP servers, driven by factors such as cryptographic operations, UDP packet handling, and the management of internal QUIC states [9]. Under high-bandwidth conditions, QUIC's performance can lag behind TCP due to its processing overhead [17]. Various techniques have been proposed to mitigate these inefficiencies, such as NIC offload [15], Client side's UDP GRO [17] and delayed ACK [6].

Despite these efforts, insufficient attention has been devoted to reducing server-side processing overhead through system call batching. While the cost of individual system calls might appear negligible, the cumulative impact becomes significant on high-performance servers that execute numerous calls [8]. io_uring, an asynchronous interface to the Linux kernel, enables the submission of multiple I/O requests with a single system call, thereby reducing context switches [5]. This makes io_uring a promising approach for reducing the syscall overhead in QUIC servers, potentially enhancing its performance.

## 1.2 Task and Goals

This project aims to integrate io_uring into the QUIC network stack to minimize the number of syscalls and context switches performed by the server, thereby enhancing its performance. Experiments are conducted, and various metrics are measured to analyze the impact of io_uring on server efficiency.

## 1.3 Overview

This following chapters are organized as follows: Chapter 2 provides an introduction to QUIC, io_uring, and related works. Chapter 3 details the design of the QUIC io_uring server, the baseline server, and the client-side implementation. Chapter 4 presents and analyzes the experimental results. Finally, Chapter 5 discusses the project's limitations and outlines potential directions for future work.

# Chapter 2

# Background and Related Work

## 2.1 Background

### 2.1.1 QUIC

Initially proposed and developed by Google [9], QUIC was later standardized by the IETF [7]. It operates in user space with enforced encryption, featuring a combined cryptographic and transport handshake that enables 0-RTT or 1-RTT connection setup [9]. Its stream-multiplexing design prevents one stream's blocking from affecting others, thereby addressing the Head-of-Line (HOL) blocking problem. It also owns an enhanced loss recovery mechanisms [9]. The user-space implementation helps QUIC overcome protocol ossification and allows for easier deployment without operating system updates. For short flow, QUIC demonstrates significant latency reduction [16]. To summarize, QUIC meets critical goals such as enhanced security, faster connection setup, reduced HOL blocking delays, and simplified deployment [11].

However, these advantages come at the cost of higher processing overhead compared to TCP. For instance, in some QUIC implementations, cryptographic operations can consume more than 40% of the total CPU time [15]. In addition, QUIC's user-space state management, including payload processing, timeout detection, and ACK generation, leads to increased system calls, context switches, and data copying between user and kernel space. Packet I/O operations alone can account for up to 50% of CPU usage [15]. This high CPU usage limits QUIC's performance under high-speed network conditions, where its data rate can be reduced by as much as 45.2% compared to TCP [17]. Some methods have been proposed to address these performance issues, some of which will be discussed in Section 2.2.

### 2.1.2 io_uring

Introduced in 2019 and available from Linux kernel version 5.1 [3], io_uring revolutionizes Linux I/O by enabling efficient, low-overhead communication between user and kernel space. Its key feature is the ability to batch multiple I/O requests using shared buffers—the submission queue (SQ) and completion queue (CQ)—which significantly reduces the number of system calls required for I/O operations.

Unlike traditional synchronous and asynchronous programming interfaces that rely on at least one system call per request [8], io_uring allows applications to queue multiple requests in the SQ and submit them all at once with a single system call (`io_uring_enter(2)`). This design reduces syscall overhead, minimizes context switches, and enhances server performance, particularly for workloads with high I/O intensity.

## 2.2 Related Work

Previous works have proposed several methods to mitigate QUIC's high processing overhead. These methods can be categorized based on the three major sources of QUIC's CPU cost, as identified in [9].

**Source #1: Cryptography**: Langley et al. [9] employ a hand-optimized version of the ChaCha20 cipher for mobile clients, aiming to reduce cryptographic overhead. Yang et al. [15] explore offloading the cryptographic module to the NIC, thereby reducing cryptographic processing in user space.

**Source #2: Sending and Receiving UDP Packets**: Zhang et al. [17] propose the use of UDP Generic Segmentation Offload (GSO) and Generic Receive Offload (GRO) to reduce the number of packets processed by the network stack. Kernel-bypass techniques, such as those described by Rizzo [10], help avoid costly data copies between user and kernel space.

**Source #3: Internal QUIC State Management**: Langley et al. [9] optimize critical paths and data structures for better cache efficiency. Iyengar et al. [6] suggest the use of delayed acknowledgments to reduce the overhead associated with frequent ACK generation.

Several proposed methods aim to reduce QUIC's CPU cost by minimizing the number of operations. For example, UDP GSO/GRO [17] and delayed ACKs [6] focus on decreasing the frequency of certain operations. Similarly, io_uring, by batching system calls, reduces the number of context switches and improves efficiencies. Based on its operational focus, io_uring should be categorized under the second source of QUIC's CPU cost: **Sending and Receiving UDP Packets**.

# Chapter 3

# Design

This chapter outlines the design of the QUIC io_uring server and the baseline server. Additionally, it introduces the client-side implementation, which simulates multiple simultaneous client connections.

Rust was chosen as the programming language, with `Quiche` [2] serving as the QUIC library and `io_uring` [12] as the low-level Linux kernel interface. ByteDance has developed `monoio` [1], a Rust runtime that leverages `io_uring`, `epoll`, and `kqueue`. However, it does not offer the low-level APIs required to implement the logic for batching syscalls. Thus, we do not use it in this project.

## 3.1 QUIC io_uring Server Design

### 3.1.1 Server Architecture

The server is designed to handle multiple client connections simultaneously. The server architecture is illustrated in Figure 3.1[1][2], which consists of a single receiver thread and eight worker threads. The receiver thread is responsible for receiving packets and delivering them to the appropriate worker threads. Each worker thread handles multiple client connections, performing tasks including packet processing, maintaining QUIC connection states, generating responses, and sending packets. Each receiver/worker thread has one dedicated ring for receiving/sending.

The data flow for a client to request data from the server are illustrated in Figure 3.1. The client begins by sending a request to the server. A receiver thread retrieves the packet from the socket and forwards it to a worker thread. The worker thread processes the request and sends a response back to the client through the same socket used for receiving packets.

Unlike TCP connections, the QUIC server does not allocate a new port for each incoming connection; instead, a single port is used for all the connections. For this reason, we do not allocate a new thread for each connection to handle both receiving and sending operations. Doing so could result in a thread inadvertently receiving data from a connection managed by another thread, which would significantly increase system complexity. This project primarily focuses on scenarios where the clients request data from the server. In this case, receiving is not a bottleneck, and a single receiver thread is sufficient to handle the workload. However, this setting may present challenges in other scenarios, such as when clients upload large amount of data to the server. This limitation will be discussed in Chapter 5.

[1]The io_uring symbol is sourced from `https://www.reddit.com/r/linux/comments/1cv9hg1/whats_new_with_io_uring_in_610/`

[2]Icons are sourced from `https://www.flaticon.com`

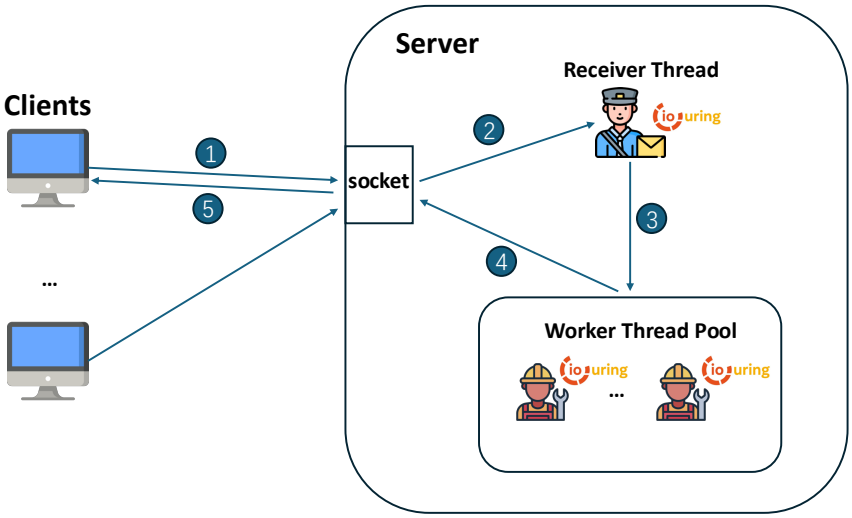



Figure 3.1: Server Architecture Design (one receiver thread and eight worker threads)

### 3.1.2 Receiver thread

The receiver thread is responsible for receiving data from the socket and forwarding it to the appropriate worker thread. To simplify the implementation, client IP addresses are used to differentiate between connections instead of relying on connection IDs. As a result, the receiver thread does not parse the packets but focuses solely on forwarding them.

For the packets reception, we enable the **multi-shot** feature. In the standard single-shot mode, a separate submission queue entry (SQE) must be submitted for each packet to receive, which corresponds to one completion queue entry (CQE). However, with multi-shot receive enabled, only a single SQE with the `RecvMsgMulti` operation needs to be submitted initially. The kernel will then fill CQEs as packets arrive. For this operation, a buffer pool must be registered, from which the kernel can get buffers for receiving packets. To achieve this, we use the Rust-based `io_uring_buf_ring` [4], with modifications to enable buffer delivery to the worker threads via a channel.

The workflow of the receiver thread is shown in figure 3.2. The receiver thread first retrieves all incoming packets from the socket by iterating through the completion queue (CQ). The received buffer is then forwarded to the corresponding worker thread via a Rust channel. Before returning to step 1 and starting the next iteration, the receiver thread attempts to recollect buffers that have been delivered back from the worker threads.

### 3.1.3 Worker thread

The worker thread is responsible for parsing packets, maintaining QUIC states, and generating/sending responses. For sending packets, io_uring is employed to batch system calls, reducing overhead and improving efficiency. The workflow of the worker thread is illustrated in figure 3.3,

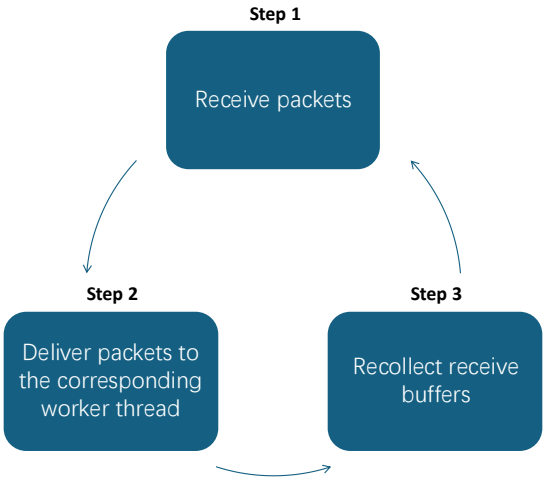



Figure 3.2: Server receiver thread workflow

which modifies the original server logic in Quiche [2] to integrate io_uring.

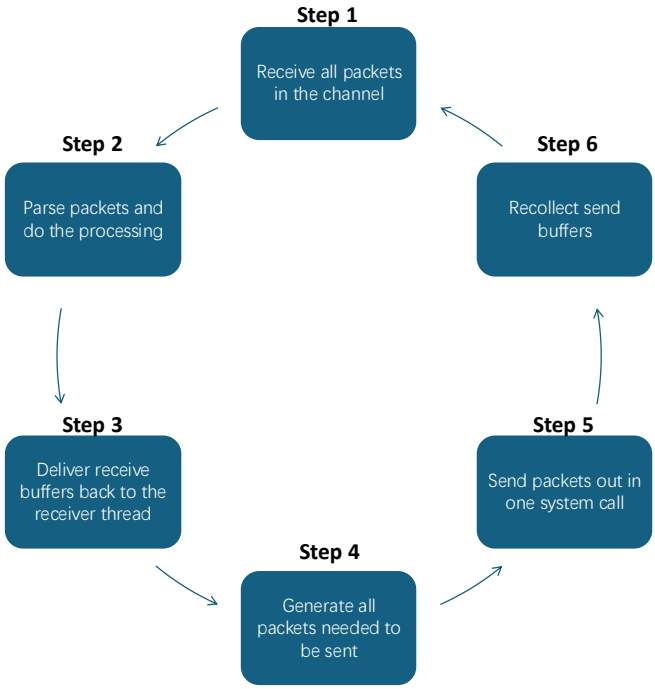



Figure 3.3: Server worker thread workflow

The key steps where io_uring is working are steps 5 and 6. In step 5, instead of invoking a system call for each packet to send, batching is performed by submitting multiple SQEs in a single

`io_uring_enter(2)` system call. In step 6, the server recollects the send buffers. However, since io_uring is an asynchronous interface, the send buffer can not be reused immediately after the system call is invoked. Instead, the server must iterate through the completion queue (CQ) to identify which send operations have completed and then safely recollect the corresponding send buffers.

## 3.2 Baseline Server Design

The only difference between the QUIC io_uring server and the baseline server is the interface used to send and receive packets. The baseline server relies on the `send_to` and `recv_from` APIs provided by the standard library. Unlike the io_uring server, which batches packet sends into a single system call, the baseline server requires one `send_to`/`recv_from` for each packet sent/received.

## 3.3 Client Design

We primarily follow the client logic in the `Quiche` library, but here we aim to handle multiple connections simultaneously. While using one thread per client might seem reasonable, system resources are limited. For I/O-intensive tasks like this, the `tokio` [13] asynchronous runtime offers a more suitable solution. A fixed number of kernel threads are launched, with each client being assigned a `tokio` task. The `tokio` task running on a thread yields control to other tasks whenever a blocking operation is executed. Once the blocking operation is completed, the task is pushed to a queue, awaiting reassignment to a thread. This design prevents the blocking of an entire thread, allowing for high concurrency and simulating multiple simultaneous client connections.

# Chapter 4

# Evaluation

This chapter begins by introducing the experimental setup and the measurements conducted. Following that, the results are presented, compared, and analyzed.

## 4.1 Experiment setup

The experiment is conducted on a single virtual machine (VM). The VM runs Ubuntu 22.04.5 LTS with kernel version 6.8.0, which supports io_uring. Additional details about the VM are provided in Table 4.1.

| | |
|---|---|
| **Ubuntu Version** | 22.04.5 LTS |
| **Kernel Version** | 6.8.0-50-generic |
| **CPU Model** | Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz |
| **CPU Architecture** | x86_64 |
| **Num of CPU(s)** | 6 |
| **Num of socket(s)** | 6 |
| **Core(s) per socket** | 1 |
| **Main Memory** | 31 GiB |

Table 4.1: VM information

To minimize interference between the server and client, the `taskset` command is used to pin the server to cores 0 and 1, while the client is pinned to cores 2, 3, 4, and 5. The kernel socket receive and send buffer sizes are set to 128 MB to prevent packet loss caused by excessive data rates.

All QUIC configurations, shown in Table 4.2, are identical for both the QUIC io_uring server and the baseline server.

The experiment is designed to be simple and straightforward. The client sends a request for an object to the server, and after the connection is established, the server responds by sending the object to the client. The time-to-last-byte (TTLB) is measured, which represents the duration between the client's request and the reception of the last byte. The requested object size is fixed at 1, 10, 50 and 80 MB, and the number of clients varies from 1 to 51. Additionally, the bandwidth on the client side is measured by maintaining a steady data stream over an extended period. To analyze system behavior, `strace` is used to record the system calls made by the server.

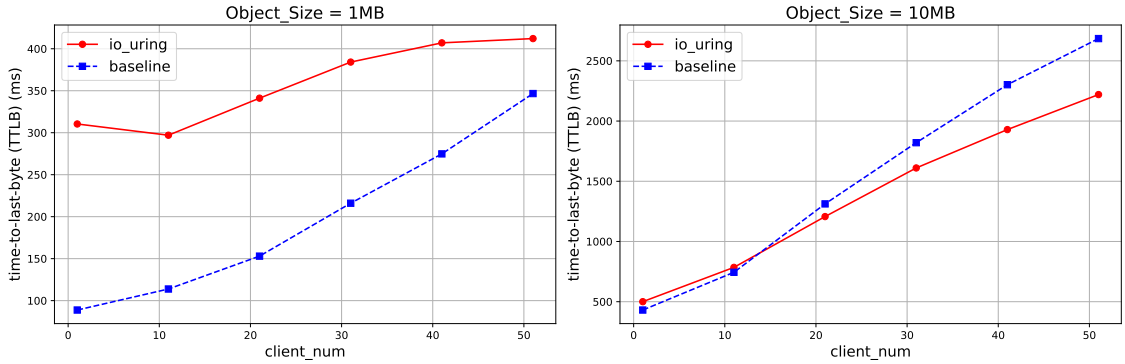
| | |
|---|---|
| **Maximum send/recv UDP Payload size (bytes)** | 1350 |
| **Maximum Idle Timeout (s)** | 10 |
| **Initial Connection-level Flow Control Window Size (bytes)** | 10,000,000 |
| **Initial Stream-level Flow Control Window Size (bytes)** | 1,000,000 |
| **Maximum Connection-level Flow Control Window Size (MB)** | 24 |
| **Maximum Stream-level Flow Control Window Size (MB)** | 16 |
| **Maximum Acknowledgment Delay (ms)** | 25 |
| **Congestion Control Algorithm** | cubic |
| **Initial Congestion Window Size (packets)** | 10 |

Table 4.2: QUIC configurations

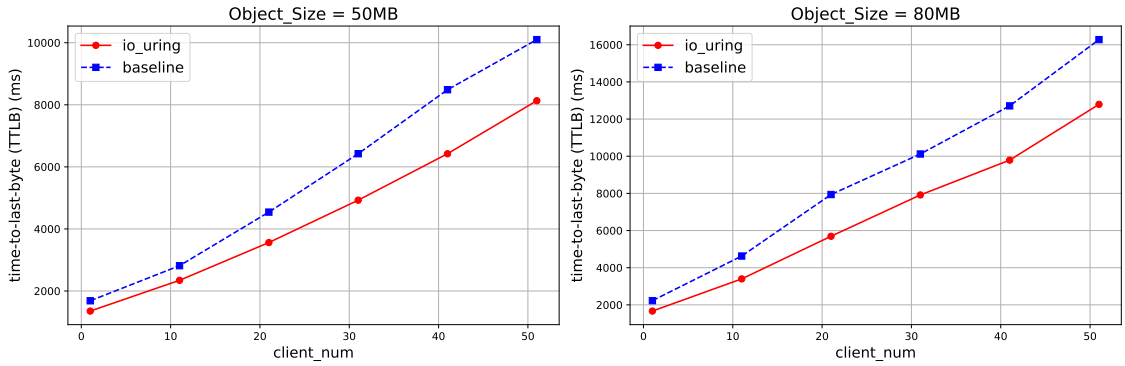
## 4.2 Result

### 4.2.1 Time to Last Byte (TTLB)

The TTLB can also be interpreted as the time required to download the entire object. For the experiments, the number of clients is set to 1, 11, 21, 31, ..., 51, and the requested object size is configured to 1MB, 10MB, 50 MB and 80 MB. We repeat the test for 5 times. Results are shown in Figure 4.1.




(a) TTLB (object size=1MB ,10MB)




(b) TTLB (object size=50MB ,80MB)

Figure 4.1: Time to Last Byte (TTLB)

For longer data flow, the QUIC server with the io_uring network stack demonstrates a shorter TTLB compared to the baseline. When the data flow size is 50 MB, the QUIC io_uring server reduces the TTLB by 16.7% with 11 clients and by 24.3% with 41 clients. On average, the io_uring network stack reduces the TTLB by 19.5% for a 50 MB data flow and by 24.4% for an 80 MB data flow.

However, for short flow—such as 1 MB in this example—the QUIC io_uring server exhibits a significantly larger TTLB. We speculate that for a short flow, the TTLB is dominated by connection setup time, during which few packets are exchanged, limiting the batching benefits of io_uring. Other inherent limitations of the server should also contribute to this issue. These drawbacks are included as part of our work's limitations in Chapter 5.

### 4.2.2 Bandwidth

To measure client-side bandwidth, the number of clients is set to 1, and the data stream is maintained steadily over an extended period. The test is repeated 10 times, and the results are presented in Figure 4.2. The average bandwidth achieved by the baseline server is 36.3 MB/s, while the QUIC io_uring server maintains an average bandwidth of 46.4 MB/s. This represents a 27.8% increase in bandwidth with the io_uring network stack.

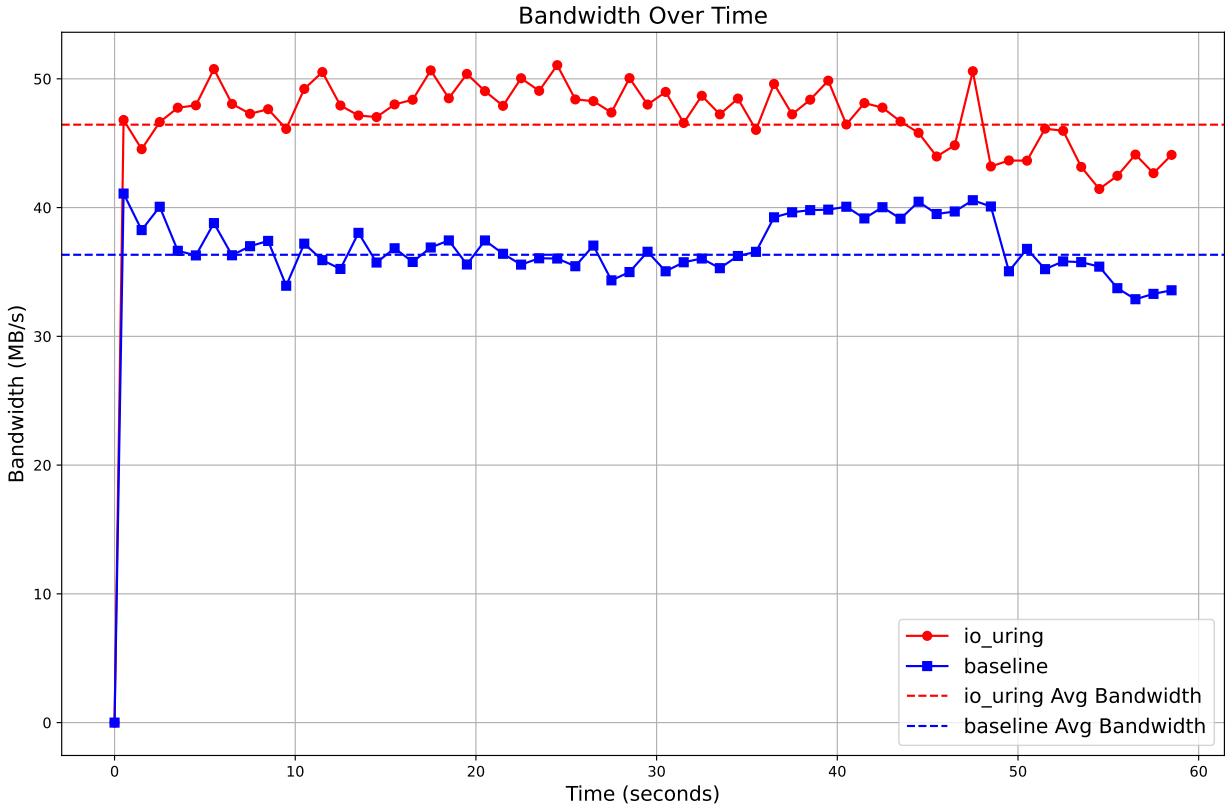



Figure 4.2: Client Side Bandwidth

### 4.2.3 System Calls

To analyze system call behavior, `strace` is used to record the system calls made by the server. Since tracing multiple threads incurs a significant performance penalty, the server is restricted to a single thread for this experiment. This thread handles receiving, processing, and sending operations. Additionally, due to the processing overhead introduced by `strace`, the test is conducted with 1 to 8 clients. However, the results are sufficient to demonstrate the batching effect of io_uring. The test is repeated 5 times.

The case with 2 clients is used to highlight the system calls made by the baseline server (Table 4.3) and the io_uring server (Table 4.4).

| Syscall | Calls | % |
|---|---|---|
| **epoll_wait** | 77.8 | 0.032 |
| **recvfrom** | 80190.6 | 33.27 |
| **sendto** | 80503.8 | 33.40 |
| **getsockname** | 80114.8 | 33.24 |
| **munmap** | 4.4 | 0.0018 |
| **write** | 2.0 | 0.00083 |
| **brk** | 144.2 | 0.060 |
| **mmap** | 3.0 | 0.0013 |
| **mremap** | 0.6 | 0.00025 |
| **Total** | 241041.2 | 100 |

Table 4.3: Syscalls made by the Quic baseline server (2 clients, data flow size = 50MB)

| Syscall | Calls | % |
|---|---|---|
| **io_uring_enter** | 50.2 | 8.48 |
| **getsockname** | 353.2 | 59.67 |
| **munmap** | 5.4 | 0.91 |
| **write** | 1.4 | 0.24 |
| **brk** | 163.4 | 27.60 |
| **mmap** | 6.6 | 1.11 |
| **mremap** | 11.8 | 1.99 |
| **Total** | 592.0 | 100 |

Table 4.4: Syscalls made by the Quic io_uring server (2 clients, data flow size = 50MB)

Figure 4.3 illustrates the total number of system calls made by the server as the number of connections increases. The results indicate that the QUIC io_uring network stack significantly reduces the number of system calls.

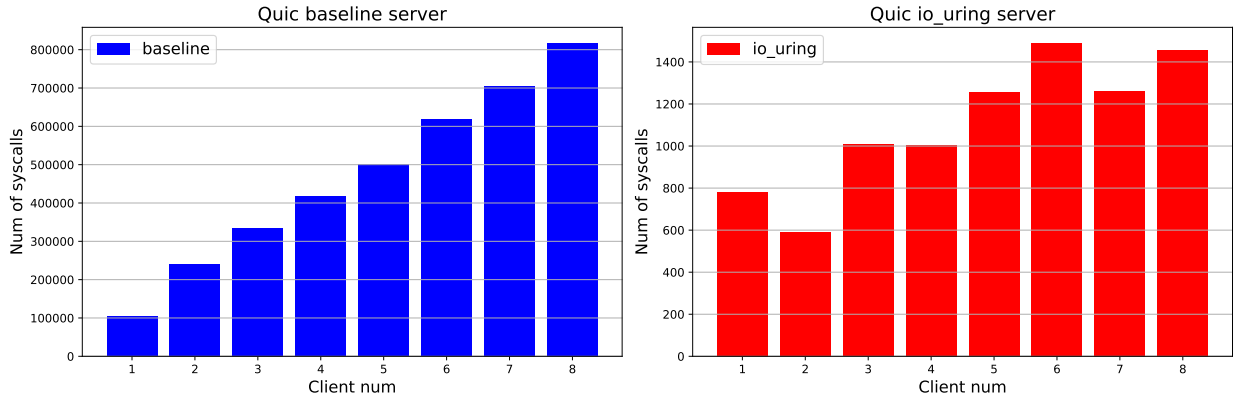



Figure 4.3: Server syscalls

To further investigate the relationship between the number of connections and the system calls, we examine whether the increase in system calls is proportional (linear), less than proportional (sub-linear), or more than proportional (super-linear). For this analysis, the number of system calls is normalized with respect to the single-client case. Figure 4.4 reveals that the baseline server's

system calls increase nearly linearly with the number of connections, while the QUIC io_uring server exhibits a sub-linear increase, which is much slower.

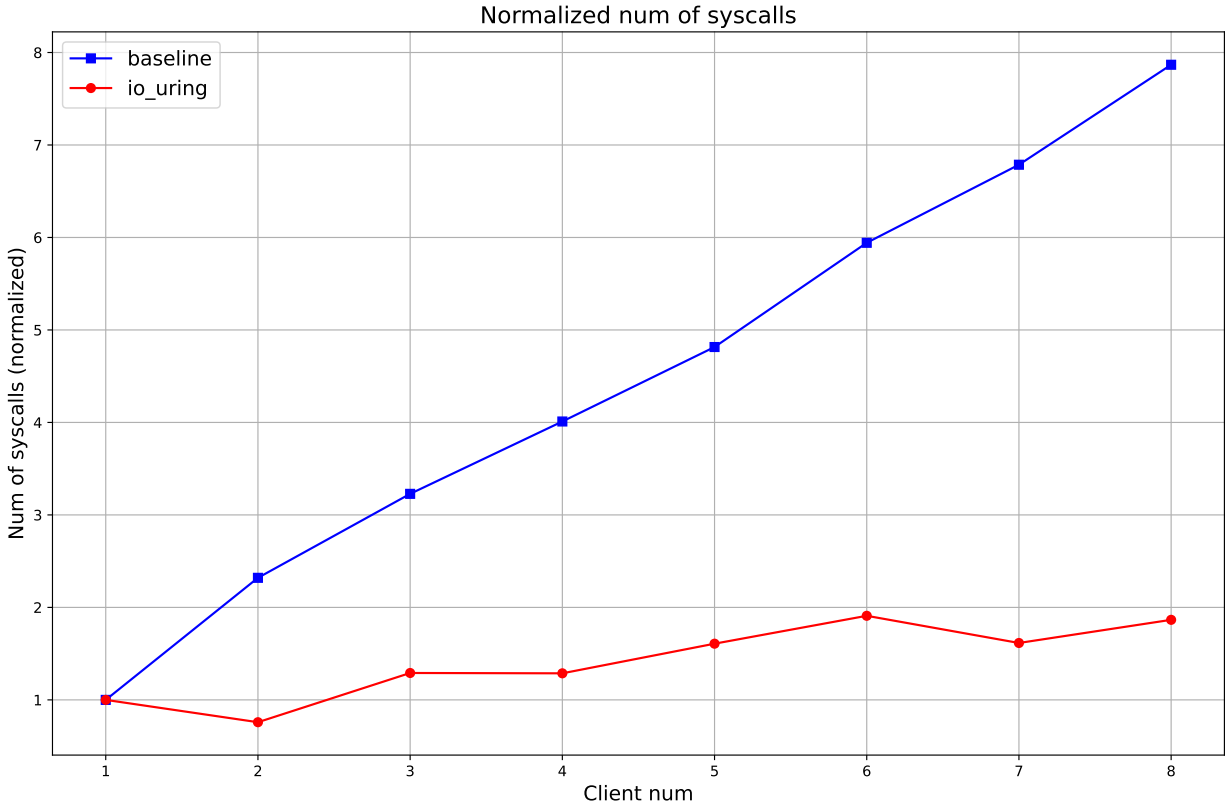



Figure 4.4: Server syscalls (Normalized)

# Chapter 5

# Outlook

In the previous chapters, we have demonstrated the performance improvements enabled by io_uring. It reduces the number of system calls, decreases download time, and helps maintain higher bandwidth. In this chapter, we discuss the limitations of our work and suggest directions for future research.

**Limitation #1: Slow Connection Establishment.** As illustrated in Figure 4.1, for a short flow, the TTLB is unusually high for the QUIC io_uring server. In such cases, the TTLB is predominantly determined by the connection setup time, during which only a limited number of packets are exchanged. As a result, the batching benefits of io_uring are not fully realized. Additionally, other potential inefficiencies within the server may contribute to this unusually slow connection establishment time. Future work should investigate this issue to identify the underlying causes and propose effective solutions. Addressing this limitation would not only improve server's performance for short data flows but could also further enhance its efficiency for longer flows.

**Limitation #2: Zero-Copy Feature.** We did not enable the zero-copy feature in the QUIC network stack. Data copying between user and kernel space introduces significant overhead, which slows down the server. Future work could implement zero-copy functionality in the QUIC io_uring server to further improve performance, especially in high-bandwidth environments.

**Limitation #3: Upload Scenarios.** Our study primarily focuses on scenarios where clients request data from the server, making sending the main bottleneck. We did not conduct tests where clients upload data to the server, which would shift the bottleneck to receiving operations. In such cases, modifications to the server design may be necessary, as a single receiver and ring might not efficiently handle high-throughput data reception. Future work could investigate these scenarios and propose optimized solutions.

**Limitation #4: Network Environment Diversity.** To show the optimization effects of batching syscalls with io_uring, we condcut the experiments under a high-bandwidth and lossless network condition. We did not evaluate the QUIC io_uring server's performance in varied network environments. Future work could explore these conditions and adapt the server implementation to achieve robust performance across different network scenarios.

# Bibliography


[1] ByteDance. Monoio: A thread-per-core rust runtime with io_uring/epoll/kqueue. `https://github.com/bytedance/monoio`, 2024.

[2] Cloudflare, Inc. quiche: an implementation of the quic transport protocol and http/3 as specified by the ietf. `https://github.com/cloudflare/quiche`, 2024.

[3] Didona, D., Pfefferle, J., Ioannou, N., Metzler, B., and Trivedi, A. Understanding modern storage apis: A systematic study of libaio, spdk, and io_uring. In *Proceedings of the 15th ACM International Systems and Storage Conference (SYSTOR '22)* (Haifa, Israel, June 2022), ACM, p. 8 pages. https://doi.org/10.1145/3534056.3534945.

[4] Holo, S. io_uring_buf_ring: A rust library for io_uring buffer ring management. `https://github.com/Sherlock-Holo/io_uring_buf_ring`, 2024.

[5] Hunter, D. Why you should use io_uring for network i/o. `https://developers.redhat.com/articles/2023/04/12/why-you-should-use-iouring-network-io`, 2023. Accessed: December 10, 2024.

[6] Iyengar, J., Swett, I., and Kühlewind, M. Quic acknowledgment frequency. Tech. rep., IETF, 2024. Work in Progress.

[7] Iyengar, J., and Thomson, M. Quic: A udp-based multiplexed and secure transport. RFC 9000, RFC Editor, May 2021.

[8] Kerrisk, M. io_uring(7) — linux manual page. `https://man7.org/linux/man-pages/man7/io_uring.7.html`, 2024. Accessed: December 9, 2024.

[9] Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J., Bailey, J., Dorfman, J., Roskind, J., Kulik, J., Westin, P., Tenneti, R., Shade, R., Hamilton, R., Vasiliev, V., Chang, W.-T., and Shi, Z. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of SIGCOMM '17* (Los Angeles, CA, USA, August 2017), p. 14 pages. https://doi.org/10.1145/3098822.3098842.

[10] Rizzo, L. netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)* (Boston, MA, June 2012), USENIX Association, pp. 101–112.

[11] Roskind, J. Quic: Design document and specification rationale. `https://goo.gl/eCYF1a`, 2012.

[12] Tokio Community. Linux io uring: the low-level io_uring userspace interface for rust. `https://github.com/tokio-rs/io-uring`, 2024.

[13] Tokio Community. Tokio: an asynchronous runtime for the rust programming language. `https://github.com/tokio-rs/tokio`, 2024.

[14] w3techs. Usage statistics of quic for websites. `https://w3techs.com/technologies/details/ce-quic`, 2024. Accessed: December 9, 2024.

[15] Yang, X., Eggert, L., Ott, J., Uhlig, S., Sun, Z., and Antichi, G. Making quic quicker with nic offload. In *Workshop on Evolution, Performance, and Interoperability of QUIC (EPIQ'20)* (Virtual Event, NY, USA, August 2020), ACM, p. 7 pages. https://doi.org/10.1145/3405796.3405827.

[16] Yu, A., and Benson, T. A. Dissecting performance of production quic. In *Proceedings of the Web Conference 2021 (WWW '21)* (Ljubljana, Slovenia, April 2021), ACM, p. 12 pages. https://doi.org/10.1145/3442381.3450103.

[17] Zhang, X., Jin, S., He, Y., Hassan, A., Mao, Z. M., Qian, F., and Zhang, Z.-L. Quic is not quick enough over fast internet. In *Proceedings of the ACM Web Conference 2024 (WWW '24)* (Singapore, Singapore, May 2024), ACM, p. 10 pages. https://doi.org/10.1145/3589334.3645323.