QUIC protocol

Luca Brugnera
Dipartimento di Matematica "Tullio Levi-Civita"
Padova, Italy
luca.brugnera.1@studenti.unipd.it

***Abstract*—TCP has often been the primary choice to fulfill the need for a reliable transport protocol. It is a rather old protocol with some flaws. With the experience gained over the years, another protocol called QUIC has been developed with the aim of achieving the same reliability, but in a faster way. This document briefly describes QUIC and compares it with TCP in different scenarios.**

I. Introduction

QUIC (Quick UDP Internet Connections) is a modern transport protocol developed by Google and later standardized by the IETF [1]. Designed to improve upon traditional protocols like TCP and TLS, QUIC operates over UDP and integrates key features such as connection multiplexing, low-latency handshake, and built-in encryption. Its primary goals are to reduce latency and enhance performance for web applications. Unlike TCP, which requires a multi-step handshake and separate setup for encryption (via TLS), QUIC combines transport and cryptographic negotiation into a single round-trip exchange. This significantly reduces connection establishment time, especially in high-latency environments. QUIC also includes stream prioritization and fast connection migration, making it highly suitable for today's mobile and data-intensive internet usage. QUIC is already widely deployed, with major adoption in HTTP/3, the latest version of the Hypertext Transfer Protocol, which uses QUIC as its underlying transport layer. This protocol represents a major shift in how data is transported over the internet, aiming for faster, more reliable, and secure communication.

II. TCP

The Transmission Control Protocol (TCP) is a cornerstone of internet communication, providing reliable, ordered, and error-checked delivery of data between applications. One significant flaw in TCP is the head-of-line (HOL) blocking problem, which affects its efficiency and responsiveness, especially in modern network applications. HOL blocking occurs because TCP delivers data strictly in order. If a single packet is lost or delayed, all subsequent packets (even if they've arrived) must wait until the missing packet is retransmitted and received. This means a single lost packet can stall the delivery of an entire stream of data, leading to unnecessary delays. This behavior is particularly problematic for applications like video streaming, online gaming, or real-time communications, where timeliness is more critical than strict ordering. Another major limitation of TCP is its inability to handle IP address changes during an active connection, commonly referred to as the migration problem. TCP connections are tightly bound to the specific IP addresses and ports of both communicating endpoints. If either device changes its IP address the existing TCP connection breaks, and all ongoing data transfers are interrupted. This is because TCP sees the new IP address as a completely different endpoint, with no way to associate it with the previous session. Even if large data can be shared in slice avoiding the retransmission from scratch, at each migration a new TCP connection must be established (and hopefully TLS too), which end up in losing unnecessary time. QUIC has been developed with these flaws in mind.

III. QUIC

A. Overview

QUIC send data through QUIC Frames, which are encapsulated QUIC packets. Likely, multiple QUIC packets are aggregated in UDP datagrams. QUIC uses the stream abstraction to multiplexing the connection. Streams are pipes where data flows in an order and reliable way.

B. Connection

Each connection possesses a set of connection identifiers, each of which can identify the connection. Differently from TCP, in QUIC a connection is identified by the aforementioned IDs, and not by source and destination ip-port couples. The primary function of IDs is to ensure that changes in addressing at lower protocol layers do not cause a connection migration, requiring a new connection establishment.

C. Stream

Streams in QUIC provide a lightweight, ordered byte-stream abstraction to an application. Streams can be unidirectional or bidirectional. Unidirectional streams carry data in one direction: from the initiator of the stream to its peer. Bidirectional streams allow for data to be sent in both directions. Streams are identified by a numeric value called stream ID. A stream is closed whenever the application sent all the data. Stream frames encapsulate data sent by an application. Stream frames are a type of QUIC frames. The sender uses `ID` and `offset` attributes in stream frames to place data in order. Stream prioritization is a technique that can be used to prioritize streams, having a significant effect on application performance.

D. Connection establishment

QUIC uses a handshake process that combines encryption setup and connection establishment into a single round trip (RTT), making it much faster than TCP with TLS. In a typical 1-RTT handshake, the client sends the first packet containing both the QUIC connection request and the TLS `Client Hello`. The server responds with its cryptographic parameters, the `Server Hello`, completing the TLS handshake and establishing the connection. During this handshake both client and server defines their connection IDs. Then, endpoints can generate and securely share with each other new IDs. If the client has previously connected to the server, a 0-RTT handshake can be used, allowing data to be sent immediately with the first message. This tight integration of TLS and transport reduces latency and improves performance, especially in mobile and real-time applications.

E. Flow Control

Receivers need to limit the amount of data that they are required to buffer, in order to prevent a fast sender from overwhelming them or a malicious sender from consuming a large amount of memory. To enable a receiver to limit memory commitments for a connection, streams are flow controlled both individually and across a connection as a whole. Similarly, to limit concurrency within a connection, a QUIC endpoint controls the maximum cumulative number of streams that its peer can initiate. A receiver sets initial limits for all streams through transport parameters during the handshake. Subsequently, a receiver sends specific QUIC frames to advertise larger limits. A receiver maintains a cumulative sum of bytes received on all streams, which is used to check for violations.

F. Congestion Control

QUIC inherits congestion control algorithm from TCP. Indeed, even if multiple implementations are possible, the suggested one is similar to NewReno. A NewReno sender is in slow start any time the congestion window is below the slow start threshold. While a sender is in slow start, the congestion window increases by the number of bytes acknowledged when each acknowledgment is processed. This results in exponential growth of the congestion window. A sender reenters slow start any time the congestion window is less than the slow start threshold, which only occurs after persistent congestion is declared or due to client ip migration. The sender exits slow start and enter a recovery period when a packet is lost. On entering a recovery period, a sender sets the slow start threshold to half the value of the congestion window when loss is detected. The congestion window must be set to the reduced value. The recovery period aims to limit congestion window reduction to once per round trip. Therefore, during a recovery period, the congestion window does not change in response to new losses. A recovery period ends and the sender enters congestion avoidance when a packet sent during the recovery period is acknowledged. A sender in congestion avoidance uses an Additive Increase Multiplicative Decrease (AIMD) approach that limits the increase to the congestion window to at most one maximum datagram size for each congestion window that is acknowledged. In case of packet loss, the sender enter in the recovery mode.

G. Recovery

QUIC packets contain one or more frames. QUIC performs loss detection based on these packets, not on individual frames. For each packet that is acknowledged by the received, all frames carried in that packet are considered received. The packet is considered lost if that packet is unacknowledged when a later sent packet has been acknowledged or by timeout. QUIC uses two thresholds to determine loss. The first is a packet reordering threshold `pktReordering`. When packet `pkt` is acknowledged, then all unacknowledged packets with a number less than `pkt - pktReordering` are considered lost. The second is related to the QUIC-measured RTT interval, the waiting time `rttReordering` which is determined as a weight factor applied to the current estimated RTT interval. If the time of the most recent acknowledgment is `timeLastAcked`, then all unacknowledged packets sent before time `timeLastAcked - rttReordering` will be considered lost. For recovery, all frames in lost packets where the associated stream requires retransmission will be placed into new packets for retransmission. The lost packet itself is not retransmitted. Streams not affected by frame losses can continue to send data, mitigating the Head-of-Line (HOL) blocking issue present in TCP. Due to this, the IETF specification suggests limiting the number of streams used within a single QUIC packet. This approach reduces the impact of packet losses.

H. Considerations

QUIC improves on TCP by addressing its main limitations (such as slow handshakes, blocking of the head of line, and lack of connection migration). To achieve the aforementioned goal, QUIC uses many frames that do not carry data but add to the protocol's complexity. Implementations must find a trade-off in the use of these types of frames: overusing them makes QUIC highly responsive to network changes but leads to higher bandwidth consumption, while underusing them allows the protocol to carry more data but reduces its responsiveness.

IV. Contribution

The remainder of this document presents an experiment designed to compare data retrieval performance using the QUIC and TCP+TLS protocols across two network scenarios. Multiple experiments were run, testing the two protocols individually, simultaneously, and in parallel with themselves. From this point forward, the term "TCP" refers specifically to TCP combined with TLS (i.e., TCP+TLS). These scenarios are intended to emulate common challenges in wireless networks, including packet loss and IP address changes. Both test environments are constructed using three Docker containers. In addition to the client and server containers, a third container runs a network simulator responsible for manipulating traffic.

The simulation environment is implemented using `ns-3` [2], a discrete-event network simulator widely adopted for protocol evaluation and experimentation. `ns-3` allows programmable definition of network topologies and behaviors. The experiment uses a simple point-to-point channel with a fixed 2Mbps bandwidth. Furthermore, the channel is configured to have an initial RTT of 50ms. Traffic manipulation is achieved through the `ErrorRateModel` class, which enables control over packet loss and modification on a per-channel basis. Further implementation details are described in the scenario-specific sections below. All application-layer traffic is bridged to the simulator container using Linux network bridges, which forward packets between containers and the `ns-3` simulation interfaces. Figure 1 illustrates the system architecture, which is based on the `quic-network-simulator`[3] framework. The use of a real QUIC implementation is necessitated by the absence of a native QUIC module in `ns-3`. For the experiment `quic-go` implementation has been used [4]. It is important to note that parameters such as packet size and flow control limits are not explicitly fixed in the experiment. As such, the reported results should be interpreted as empirical observations. Nonetheless, both TCP and QUIC operate under similar congestion control regimes: TCP employs the `NewReno` algorithm, while QUIC uses a functionally equivalent NewReno-like algorithm as specified in RFC 9002[5].

A. Client

The client initiates the download of a 4MB randomly generated payload using both TCP and QUIC. This is achieved by concurrently running two distinct Go-based client applications. Due to the potential for connection disruptions during large data transfers over TCP, the client is configured to use HTTP `Range` headers to resume partially completed downloads, minimizing redundant retransmissions. QUIC, in contrast, supports seamless IP address migration via connection ID abstraction. This eliminates the need for explicit resume mechanisms and allows connection continuity across endpoint address changes.

B. Server

The server listens for incoming connections on both TCP and UDP port 4433. To support range-based requests from the TCP client, the server exposes endpoints capable of responding with byte-range content.

C. Lossy Scenario

Wireless links are prone to higher packet loss rates compared to wired links. To emulate this, a custom `ErrorRateModel` instance defines a scenario where packets are lost at a 2% rate. In this environment, both TCP and QUIC were tested using different configurations.
Initially, the protocols were tested individually, halving the bandwidth. Figure 2 shows the outcomes of the aforementioned experiment.
Subsequent experiments using this scenario restore the bandwidth to 2 Mbps, because two connections are established concurrently. Figure 3 presents the results of the simulation with two concurrent connections using the same transport protocol. The graphs highlight that both protocols are fair, as they share the medium equally.
Finally, Figure 4 illustrates TCP and QUIC connections competing against each other. In this comparison, it is clear that QUIC outperforms TCP. Nevertheless, QUIC and TCP are both friendly protocols, as they each utilize a similar portion of the medium.

Throughput is influenced by both congestion control and the loss recovery algorithm. Assuming that congestion control behaves similarly in both protocols, the results suggest that QUIC's recovery algorithm is slightly more effective than TCP's. Overall, the results indicate that QUIC outperforms TCP under packet loss, maintaining higher and more stable throughput. As discussed in Section III-G, QUIC streams not affected by losses are still able to continue exchanging data normally.

D. NAT Scenario

This scenario simulates client-side IP mobility, as often encountered in wireless environments due to client mobility. The simulation uses a modified `ErrorRateModel` to periodically change the source and destination IP addresses of packets (every 10 seconds), emulating client IP migration. As in the previous scenario, multiple experiments were run using different configurations. Each experiment with just a single active connection uses halved bandwidth. Figure 5 presents comparative results for TCP and QUIC under these conditions. The outcomes underline how, at each client IP migration, TCP actually closes and establishes a new connection, whereas QUIC simply keeps retrieving data from the server.
Figure 7 shows the outcomes of experiments where concurrent TCP or QUIC connections are established.
Finally, Figure 6 illustrates the case where TCP and QUIC compete against each other.

QUIC demonstrates superior robustness to address changes, preserving session continuity, whereas TCP connections reset on each IP reassignment, incurring throughput penalties.

E. Conclusion

The results confirm that QUIC provides significant advantages over TCP in both lossy and mobile network environments. Its built-in support for connection migration and reduced sensitivity to packet loss make it a strong candidate for deployment in wireless settings.

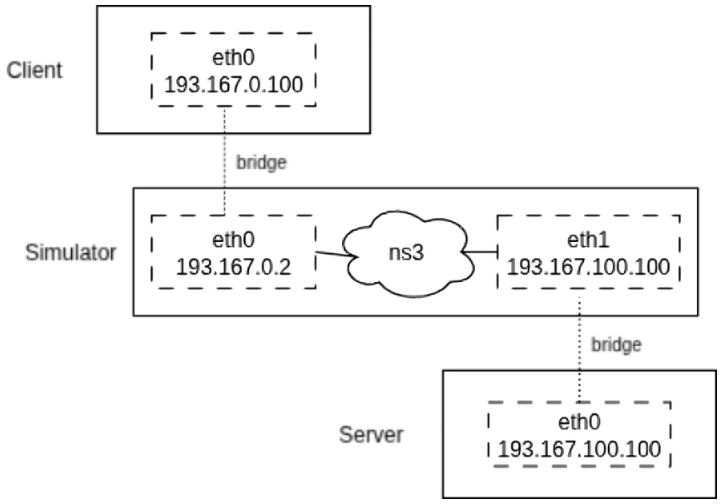

Fig. 1. System architecture based on the `quic-network-simulator` design.

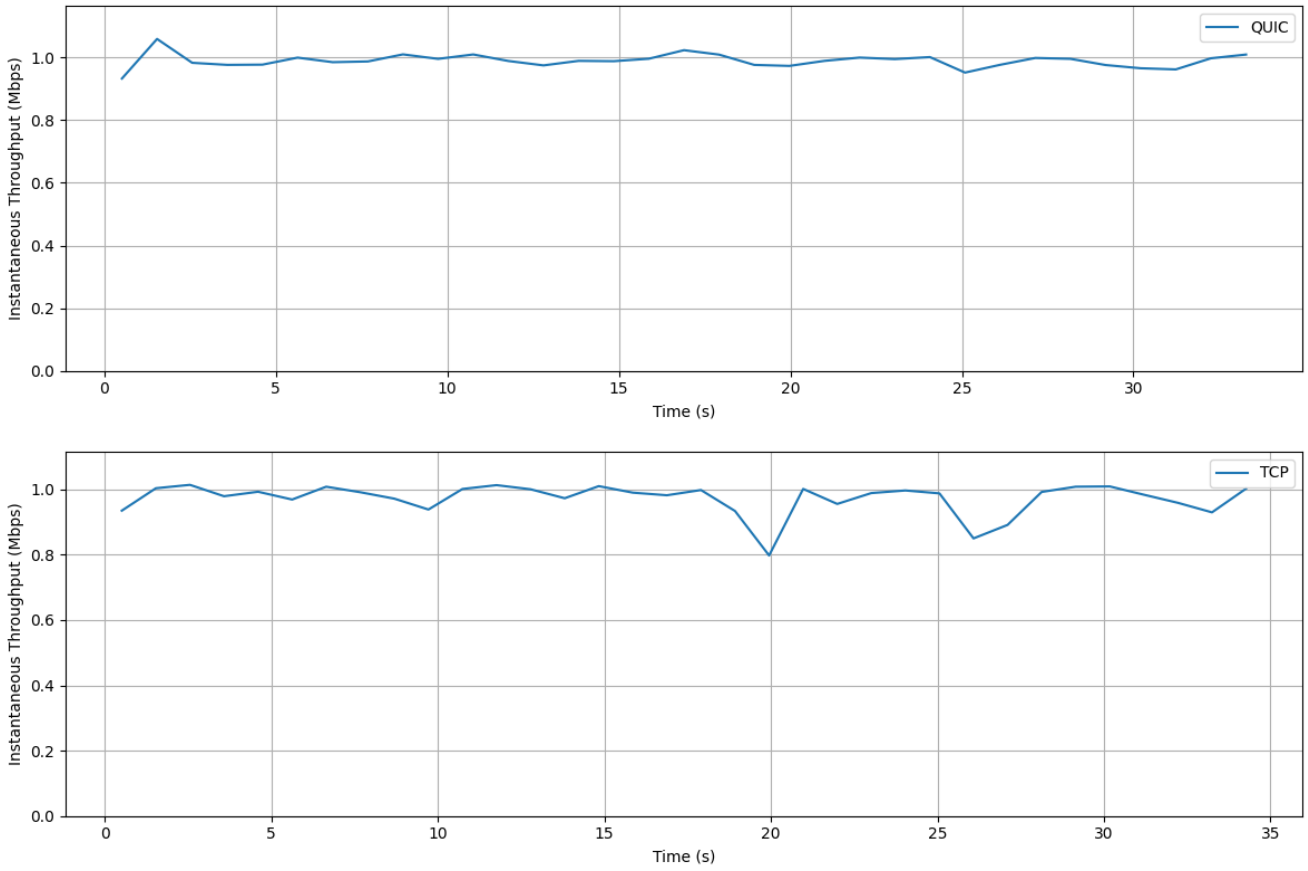

Fig. 2. QUIC or TCP single connection throughput in lossy scenario.

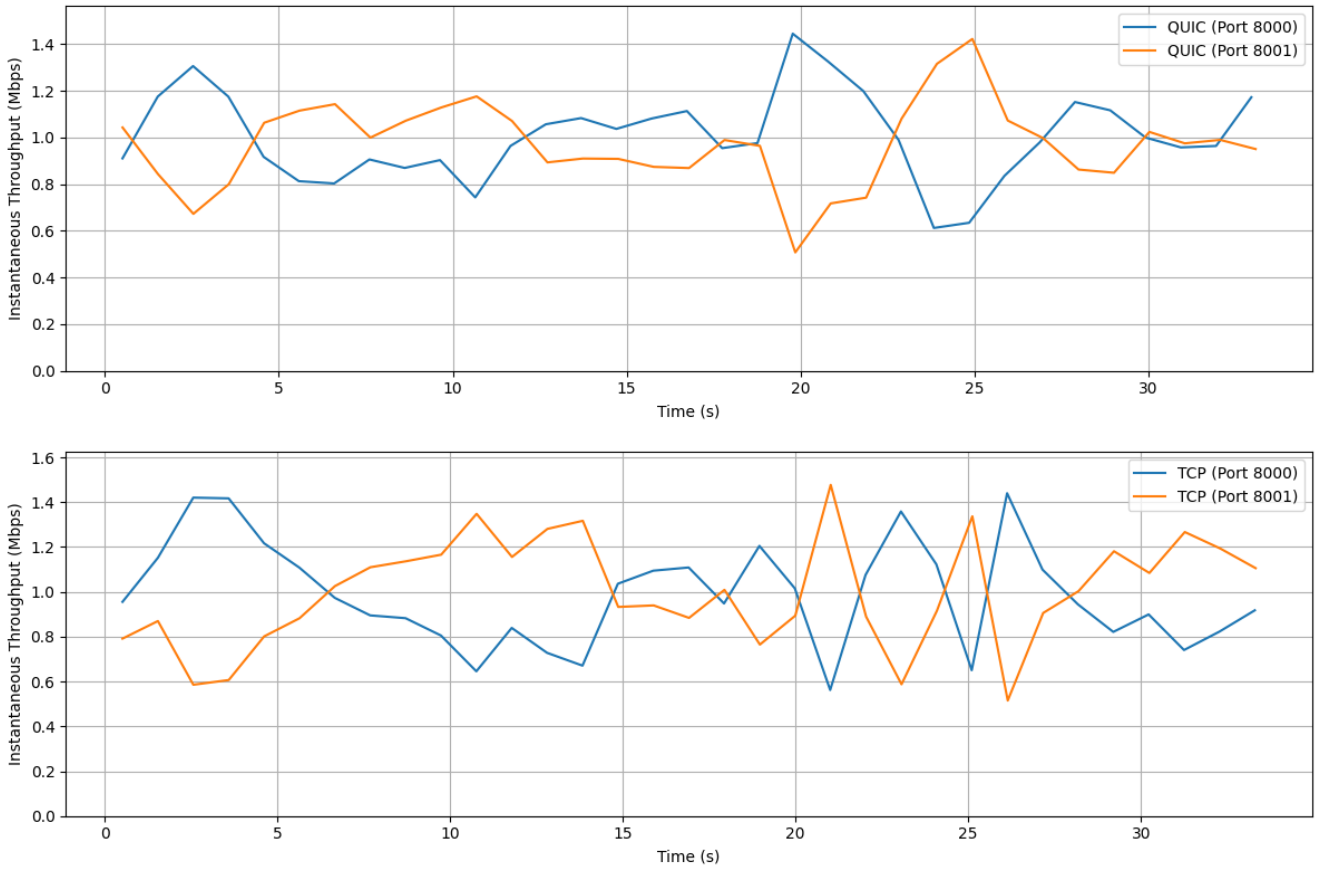

Fig. 3. Concurrent QUIC or TCP connections throughput in lossy scenario.

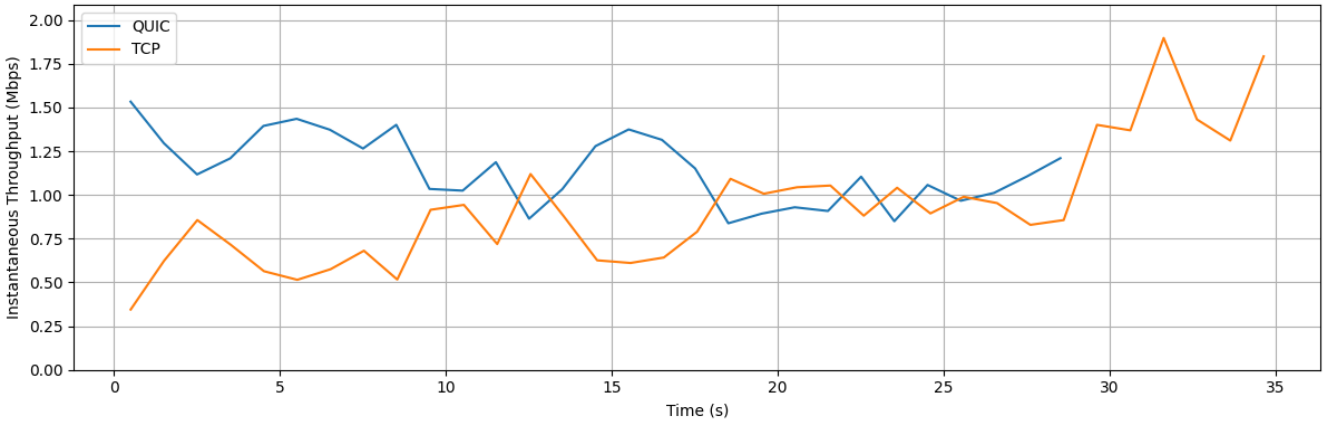

Fig. 4. Concurrent QUIC and TCP connections throughput in lossy scenario.

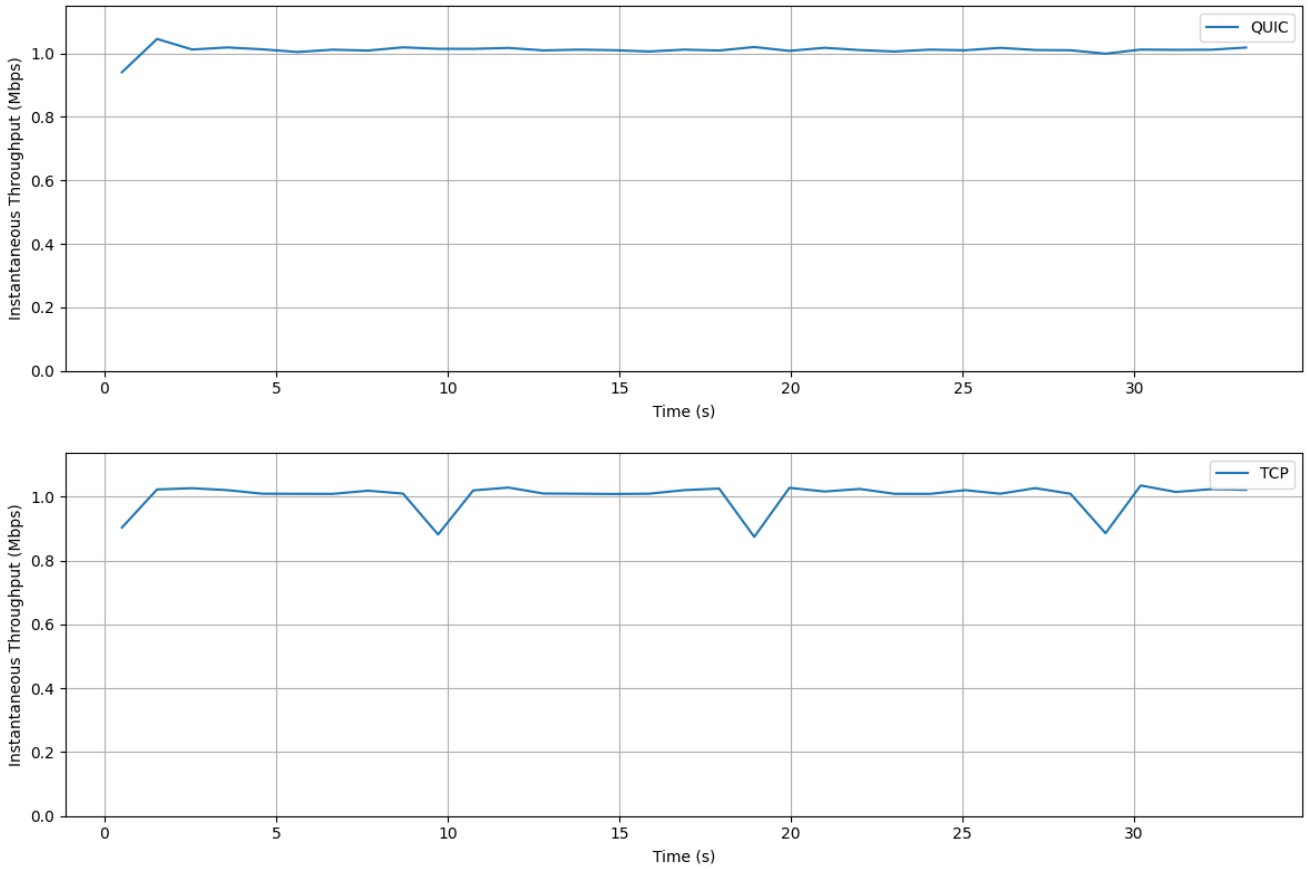

Fig. 5. QUIC or TCP single connection throughput in NAT scenario.

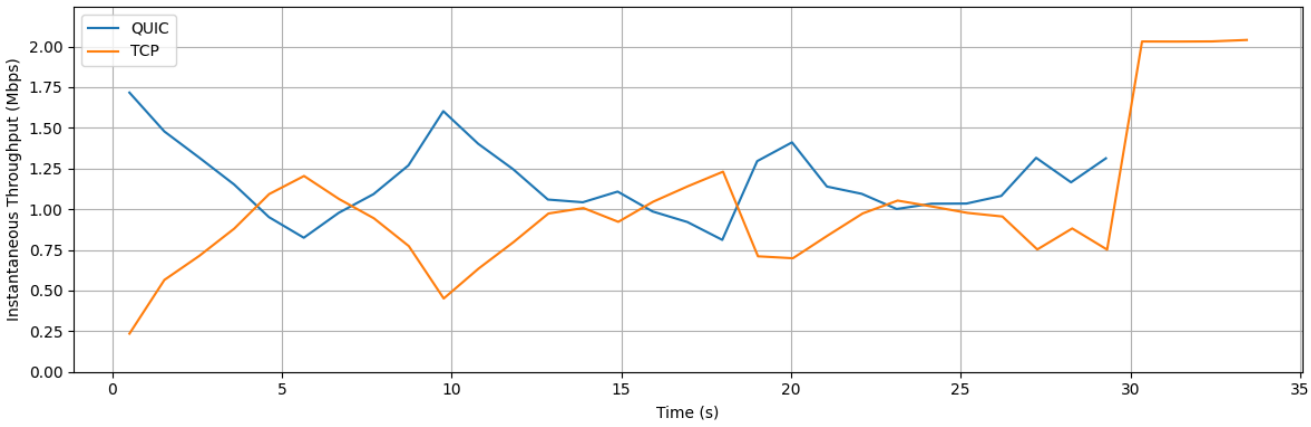

Fig. 6. Concurrent QUIC and TCP connections throughput in NAT scenario.

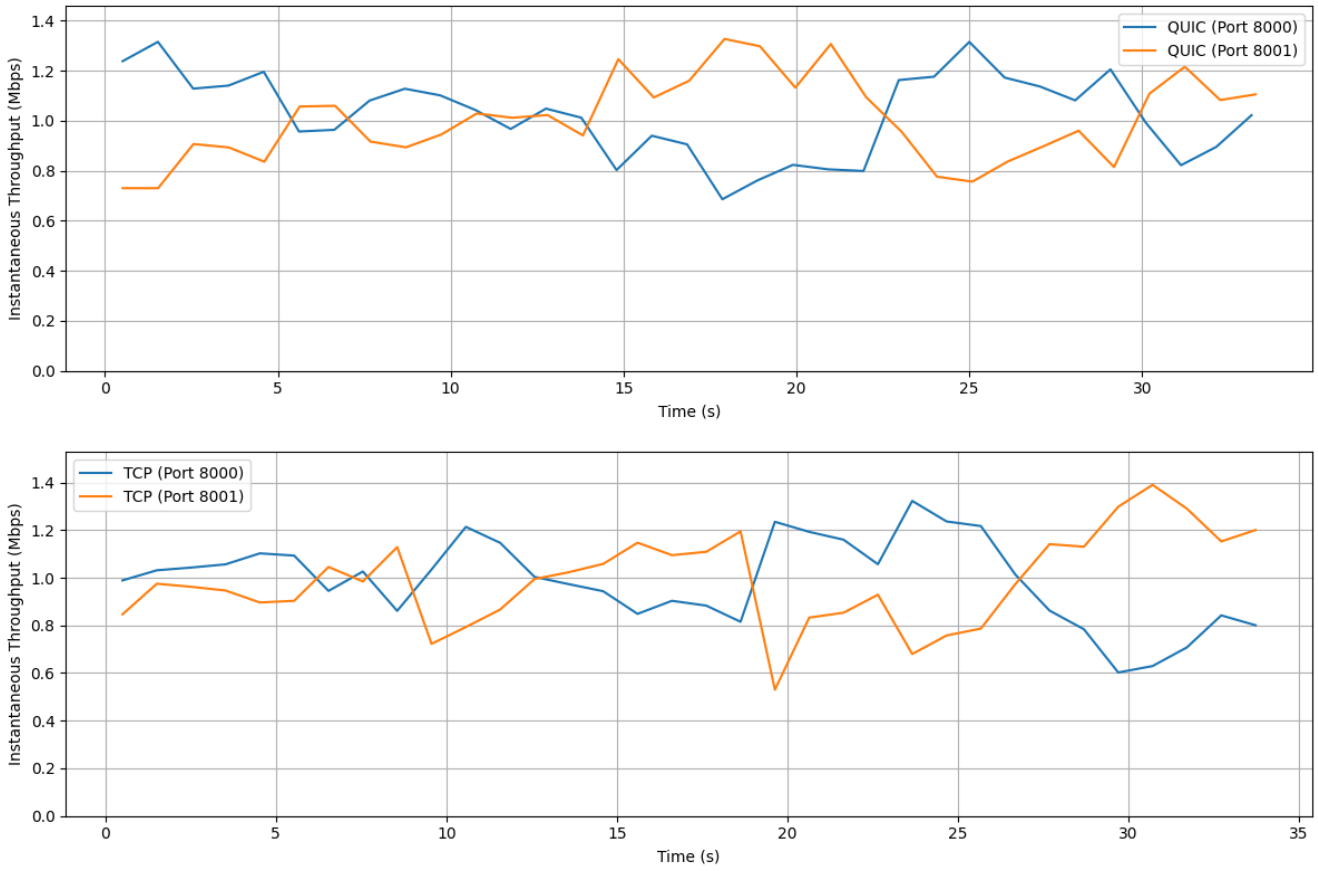

Fig. 7. Concurrent QUIC or TCP connections throughput in NAT scenario.

REFERENCES

[1] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. https://datatracker.ietf.org/doc/html/rfc9000. RFC 9000. 2021.

[2] ns-3 Project. *ns-3 Network Simulator*. https://www.nsnam.org. Official Project Website. 2025.

[3] Google QUIC Working Group. *quic-network-simulator*. https://github.com/google/quic-network-simulator. GitHub Repository. 2025.

[4] Lucas Clemente and Contributors. *quic-go: A QUIC implementation in Go*. https://github.com/quic-go/quic-go. GitHub Repository. 2025.

[5] Jana Iyengar and Ian Swett. *QUIC Loss Detection and Congestion Control*. https://datatracker.ietf.org/doc/html/rfc9002. RFC 9002. 2021.