Performance Comparison of QUIC and TCL+TLS Protocols in a File Transfer Application

Contents

1 Introduction

In a time dominated by data-driven applications, efficient and secure file transfer protocols are critical for enabling real-time communication, cloud services, and large-scale data sharing. Traditional protocols like TCP (Transmission Control Protocol) paired with TLS (Transport Layer Security) have long served as the backbone for reliable and encrypted data transmission. However, emerging technologies such as QUIC (Quick UDP Internet Connections) promise to address long standing limitations of TCP, such as latency during connection setup and head-of-line blocking, while integrating modern security features like mandatory encryption.

1.1 Motivation

While TCP + TLS remains widely adopted, its performance bottlenecks in high-latency or unstable networks have spurred interest in alternatives. QUIC, built on top of UDP, offers features like 0-RTT connection resumption, multiplexed streams, and built-in encryption with TLS 1.3, positioning it as a compelling candidate for modern applications. However, its trade-offs in terms of resource usage, compatibility, and security robustness are less understood particularly in the context of file transfer systems. This work seeks to evaluate whether QUIC's theoretical advantages translate into practical benefits for file transfer applications compared to the well-established TCP+TLS stack.

2 Background

Secure communication over the Internet has long relied on the interplay between protocols that guarantee reliability and those that ensure privacy. Historically, this has been achieved through the layered combination of TCP (Transmission Control Protocol) and TLS (Transport Layer Security) - a partnership that prioritizes backward compatibility and incremental evolution. In contrast, QUIC (Quick UDP Internet Connections) represents a paradigm shift, integrating transport and security into a unified protocol designed for the demands of modern networks. Understanding their architectural philosophies is key to evaluating their performance and security trade-offs.

2.1 The Layered Approach: TCP + TLS

TCP, the bedrock of Internet communication since the 1970s, was designed to solve a fundamental problem: how to reliably transmit data across inherently unreliable networks [1]. By establishing connections through its three-way handshake, enforcing in-order packet delivery, and dynamically adjusting transmission rates via congestion control algorithms like Cubic and BBR [2, 3], TCP ensures that data arrives intact and sequenced. However, this reliability comes at a cost. For instance, TCP's strict in-order delivery creates head-of-line blocking, where a single lost packet stalls all subsequent data—an issue that becomes more pronounced in high-latency or lossy environments like mobile networks or satellite connections.

To address TCP's lack of inherent security, TLS emerged as a cryptographic layer operating on top of it. TLS 1.3, the latest iteration, streamlines the handshake process to a single round trip (or zero for resumed connections) and employs modern encryption suites like AES-GCM and ChaCha20-Poly1305. Yet, because TLS operates as a separate layer, establishing a secure connection requires sequential handshakes: first, TCP negotiates the transport channel, then TLS authenticates and encrypts it. For a new connection, this results in 2.5 round trips of latency before any application data is transmitted - a delay that impacts global-scale systems, especially when round-trip times often exceed 100 milliseconds.

This layered design, while modular and backward-compatible, introduces inefficiencies. Middle entities like firewalls and NATs [4], optimized for decades of TCP traffic, often misinterpret or hinder TLS-specific optimizations. Moreover, TCP's head-of-line blocking persists even after TLS encrypts the data, as packet loss at the transport layer stalls the entire encrypted stream. These limitations - coupled with the growing demands of high-speed, low-latency applications - motivated the search for a protocol that could reimagine both transport and security as a whole.

2.2 QUIC

QUIC, first deployed by Google in 2012 and later standardized as the foundation of HTTP/3, challenges the traditional layered approach of TCP + TLS. Built on top of UDP, a protocol without built-in reliability

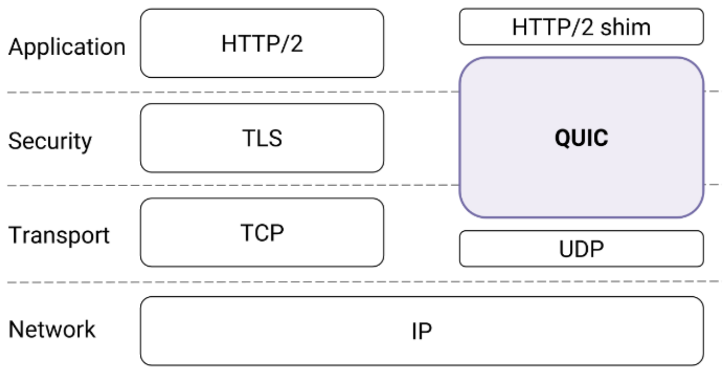


Figure 1: QUIC in the traditional HTTPS stack [4].

or congestion control, QUIC integrates transport and cryptographic handshakes into a single layer. This integration reflects a design philosophy where security is mandatory, latency is minimized, and reliability is redefined to suit the demands of modern web traffic.

From its inception, QUIC mandated encryption. Unlike TCP + TLS, where encryption is an optional add-on, QUIC embeds TLS 1.3 directly into its handshake, ensuring that every connection is secured by default. This allows QUIC to merge the transport and cryptographic handshake into a single round trip for new connections and 0-RTT for resumed sessions. For example, a user resuming a connection on a mobile network could begin transmitting data immediately, avoiding the 200+ milliseconds of handshake latency typical of the traditional stack. This performance boost is particularly noticeable in modern, latency-sensitive applications such as video streaming, online gaming, and real-time communications, where reducing latency even by a few milliseconds can significantly improve user experience.

QUIC further diverges from TCP through its handling of multiplexing. By allowing multiple independent streams within a single connection, QUIC eliminates head-of-line blocking at the application layer. If a packet is lost on one stream, other streams continue unaffected - an essential feature for real-time applications where delays and interruptions can degrade user experience. Additionally, QUIC supports connection migration, enabling seamless transitions between networks (e.g., from Wi-Fi to mobile data) without renegotiating security parameters. This flexibility is a critical advantage in mobile environments where users frequently switch between network types, a challenge that TCP's strict connection-to-IP-address binding cannot accommodate.

QUIC's efficiency and performance have led to widespread adoption, with major services like Google, YouTube, and Cloudflare embracing it as the foundation of HTTP/3. This adoption further underscores its significance in improving internet performance for a growing number of high-traffic and performance-demanding applications.

3 Methodology and Implementation

This section details the methodology used for the development and evaluation of the file transfer application implementations using both TCP+TLS and QUIC protocols, and describes the corresponding server and client implementations. Both systems were developed in Python, with careful attention to fairness in testing conditions (e.g., identical file sizes, network emulation settings).

3.1 Workflow and Functional Dynamics

The TCP+TLS implementation relies on synchronous, sequential operations. Clients initiate connections through Python's `socket` and `ssl` libraries, wrapping a TCP socket with TLS encryption after the initial handshake. File transfers are structured around JSON-formatted commands (e.g., {"action" : "send_file"}), with data split into 4 KB chunks for transmission. This approach ensures simplicity but struggles with concurrency, as each connection handles only one stream at a time.

In contrast, QUIC, powered by the `aioquic` library, embraces asynchronism. Clients and servers communicate over multiplexed streams within a single QUIC connection, enabling simultaneous uploads and downloads without head-of-line blocking. Commands are sent as plaintext (e.g., upload filename), reducing protocol overhead. The asynchronous model, managed via Python's `asyncio`, allows QUIC to handle multiple streams concurrently, making it inherently scalable for high-throughput scenarios.

Security Foundations. Both systems use self-signed RSA certificates, generated automatically, to authenticate servers and establish TLS 1.3 encryption. However, their security workflows differ subtly. In TCP+TLS, the client skips hostname verification for simplicity, accepting any server certificate – a pragmatic choice for testing, though not suitable for production environments. In contrast, QUIC enforces certificate validation by default, requiring clients to trust the server's certificate explicitly. This reflects QUIC's design philosophy, where security is mandatory rather than optional. Encryption parameters are aligned for fairness: AES-GCM secures data in transit, and elliptic-curve Diffie-Hellman ensures forward secrecy. Despite these similarities, QUIC's integration of TLS 1.3 into its handshake eliminates vulnerabilities associated with middleboxes (e.g., firewalls that mishandle TCP/TLS extensions).

3.2 Implementation Details

Python was chosen as the implementation language due to its ease of use, rich ecosystem of libraries, and rapid prototyping capabilities. Libraries such as `socket` and `ssl` provide robust support for TCP+TLS communications, while the asynchronous capabilities of `asyncio` together with `aioquic` were used for a modern implementation of the QUIC protocol. This choice enables a direct comparison between the traditional synchronous model and the modern asynchronous paradigm.

3.2.1 TCP + TLS

Server. The TCP+TLS server is implemented using Python's `socket` and `ssl` libraries. The server creates a TCP socket, binds it to a specified host and port, and then wraps the socket with an SSL context to enforce TLS encryption. The server supports multiple operations including file upload, file download, and simple ping requests. Commands are exchanged in JSON format, and file transfers are performed in 4 KB chunks to ensure manageability and facilitate the measurement of performance metrics such as throughput and transfer time.

Client. The TCP+TLS client mirrors the server's functionality. It establishes a secure connection to the server using an SSL-wrapped socket and sends JSON-formatted commands to initiate file uploads or downloads, or to perform ping tests. The client measures key performance metrics such as handshake time, round-trip time (RTT), and throughput. This implementation serves as a baseline for comparing the performance and efficiency of the traditional TCP+TLS approach against the QUIC protocol, where a similar approach was used to ensure a direct comparison.

3.2.2 QUIC

Server. The QUIC server is implemented with the `aioquic` library, which provides support for QUIC and HTTP/3 protocols. This server handles connections asynchronously, allowing multiple independent streams within a single QUIC connection. Each stream can be used to process different commands such as file upload, download, or ping, eliminating the head-of-line blocking issue inherent in TCP. The server integrates TLS 1.3 directly into the QUIC handshake, reducing connection setup time and enhancing security. Performance metrics are recorded for each operation to allow for a detailed comparison with the TCP+TLS implementation.

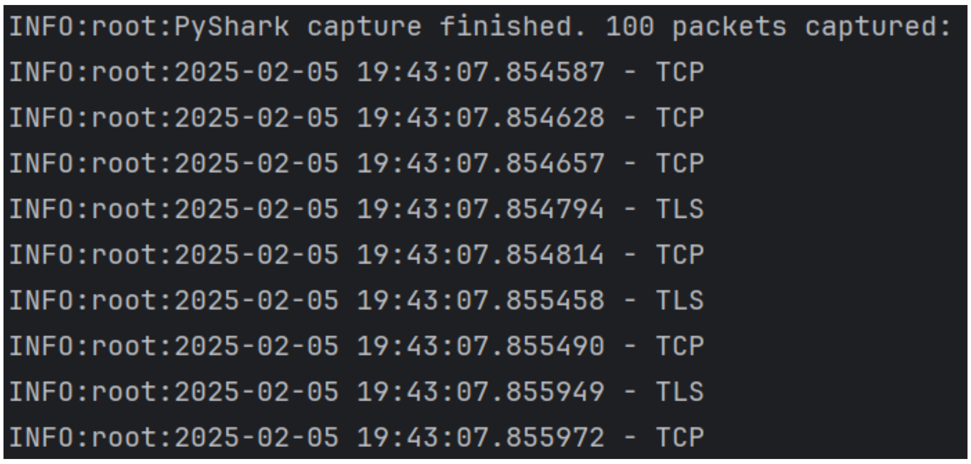
Client. The QUIC client utilizes `aioquic` and Python's `asyncio` framework to establish a secure, asynchronous connection with the server. It creates bidirectional streams for each operation (upload, download, ping) and uses plaintext commands to minimize overhead. The client captures performance metrics such as handshake duration, RTT, and throughput for each transfer. Additionally, the QUIC client supports concurrent stream processing, which is particularly beneficial for high-throughput or low-latency applications. These measurements are then compared with those obtained from the TCP+TLS implementation.

3.2.3 PyShark Analysis

PyShark is employed to capture and analyze network packets in real time, providing insights into the actual data exchanged during protocol operations. Two distinct configurations were implemented:

- **TCP+TLS:** A separate thread initiates a live capture on the loopback interface (using a filter such as `tcp.port == 8443`) to verify the sequence of the TLS handshake and file transfer operations.
- **QUIC:** The capture is configured with the filter `udp.port == 4433` to analyze QUIC traffic, observing the handling of multiple streams.

Captured data (timestamps, highest protocol layer, number of packets) is logged to support the analysis of performance and protocol behavior.



```
INFO:root:PyShark capture finished. 100 packets captured:
INFO:root:2025-02-05 19:43:07.854587 - TCP
INFO:root:2025-02-05 19:43:07.854628 - TCP
INFO:root:2025-02-05 19:43:07.854657 - TCP
INFO:root:2025-02-05 19:43:07.854794 - TLS
INFO:root:2025-02-05 19:43:07.854814 - TCP
INFO:root:2025-02-05 19:43:07.855458 - TLS
INFO:root:2025-02-05 19:43:07.855490 - TCP
INFO:root:2025-02-05 19:43:07.855949 - TLS
INFO:root:2025-02-05 19:43:07.855972 - TCP
```

Figure 2: Example of TCP+TLS packets captured using PyShark.

4 Results

In this section, we summarize the performance measurements and packet-level analysis obtained through the benchmarking routines and PyShark captures.
The performance of both implementations is evaluated in terms of:

- **Latency Metrics:** Handshake time and Round-Trip Time (RTT), including standard deviation.
- **Transfer Metrics:** Upload and download times, and calculated throughput (in MB/s).

Both the TCP+TLS and QUIC clients incorporate dedicated modules to aggregate these metrics (e.g., via the `BenchmarkStats` class) and present the results in formatted tables using the `tabulate` library.

4.1 Summary of Results

The following table summarize the average benchmark metrics obtained for each protocol. The numbers refer to the average time passed for 5 ping requests, 3 uploads and 3 downloads of a file of custom size that can be created by the user (in this case, the results refer to a file of 1GB size).

Metric	**TCP+TLS**	**QUIC**
Handshake Time	0.006017 s	0.019117 s
Average RTT	0.000950 s	0.005623 s
RTT Std. Dev.	0.000492 s	0.003769 s
Average Upload Time	4.422592 s	890.674859 s
Average Upload Throughput	246.68 MB/s	1.15 MB/s
Average Download Time	4.284258 s	885.309263 s
Average Download Throughput	254.56 MB/s	1.16 MB/s

Table 1: Comparison of network performance metrics between TCP+TLS and QUIC.

These metrics provide a detailed comparison of TCP+TLS and QUIC in terms of connection setup, latency, and data transfer performance. Handshake Time represents the time taken to establish a secure connection, which directly affects how quickly a session can begin. A lower handshake time is preferable, especially for applications requiring fast setup, such as real-time communications. Average RTT (Round-Trip Time) measures the time it takes for a packet to travel to its destination and back, serving as an indicator of network responsiveness. A lower RTT suggests better performance, particularly for latency-sensitive applications like video calls or online gaming. RTT Standard Deviation quantifies the variability in RTT, providing insight into network stability; high variability can lead to inconsistent performance and jitter. Average Upload and Download Time indicate how long it takes to transfer data in either direction, which is critical for evaluating protocol efficiency under different network conditions. Average Upload and Download Throughput measure the rate at which data is successfully transmitted, with higher values indicating more efficient use of available bandwidth.

4.2 Detailed Results

The following tables present the download and upload performance for both TCP+TLS and QUIC, focusing on two key metrics: transfer time and throughput. Transfer time indicates how long it takes to complete the data transfer, while throughput measures the rate at which data is successfully transmitted. Lower transfer times and higher throughput values suggest better performance. Each test was repeated three times to ensure accuracy and minimize the impact of network fluctuations. The results for upload performance are presented first, followed by download performance.

Upload Results. We tested the upload performance over a 1GB test file (`test_upload.bin`), which was generated by the code itself. The results below show the download time and throughput for both TCP+TLS and QUIC across three iterations.

Iteration	**TCP + TLS**		**QUIC**	
	Time (s)	Throughput (MB/s)	Time (s)	Throughput (MB/s)
1	3.582199	285.86	894.224662	1.15
2	3.571778	286.69	889.082609	1.15
3	6.113801	167.49	888.717308	1.15

Table 2: Performance comparison of upload time and throughput between TCP+TLS and QUIC across three iterations.

Download Results. For the upload test, we followed a similar approach, using a 1GB test file (`test_download.bin`) generated by the code itself. The table below shows the upload time and throughput for both TCP+TLS and QUIC across three iterations.

Iteration	**TCP + TLS**		**QUIC**	
	Time (s)	Throughput (MB/s)	Time (s)	Throughput (MB/s)
1	5.917546	173.04	887.140160	1.15
2	3.449762	296.83	885.737737	1.16
3	3.485467	293.79	883.049894	1.16

Table 3: Performance comparison of download time and throughput between TCP+TLS and QUIC across three iterations.

In both protocols, each iteration of download and upload resulted in a slight variations in transfer time and throughput across iterations. As can be clearly seen, QUIC implementation showed significantly higher transfer times and much lower throughput compared to the TCP+TLS implementation, which is attributed to a bottleneck somewhere in the server implementation, probably due to the fact that the implementation of the QUIC protocol in Python is still immature.

4.3 PyShark-based Traffic Analysis

For both protocols, PyShark was used to:

- Verify the sequence of handshake messages.
- Monitor the data transfer phases.
- Detect anomalies such as retransmissions or unexpected protocol behavior.

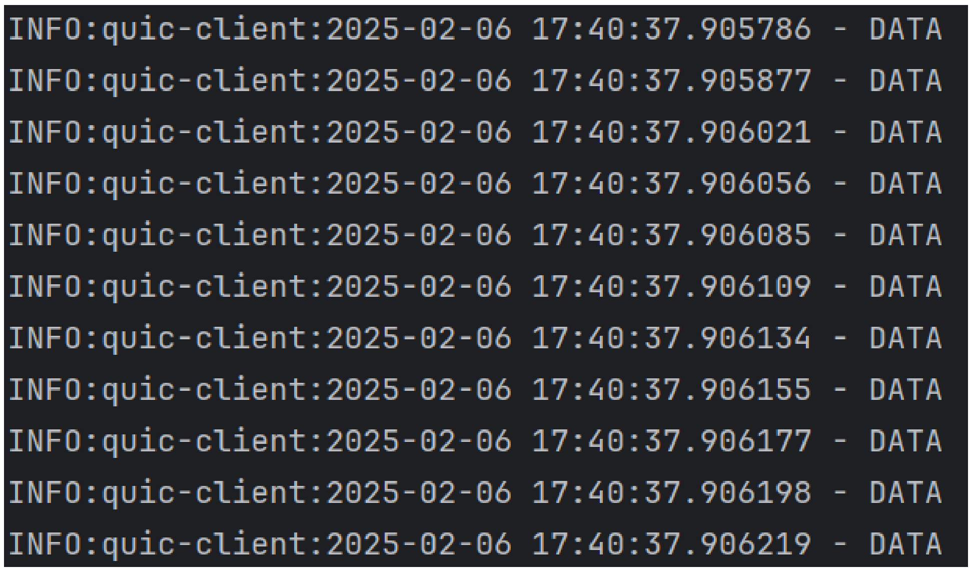
Separate captures were configured for TCP+TLS (filtering on `tcp.port == 8443`) and for QUIC (filtering on `udp.port == 4433`). The results provide a complementary view to the numerical performance metrics.

TCP+TLS PyShark Observations:

- Ping operations captured 2 packets.
- File transfer operations (upload/download) captured approximately 98 packets, confirming the expected segmentation and sequence. Possibly due to the implementation, the packet captured were always 2 short of the number of packets set.

QUIC PyShark Observations:

- Ping operations captured 4 packets.
- Upload and download operations captured 100 packets each, with packet labels indicating encrypted QUIC DATA frames.



```
INFO:quic-client:2025-02-06 17:40:37.905786 - DATA
INFO:quic-client:2025-02-06 17:40:37.905877 - DATA
INFO:quic-client:2025-02-06 17:40:37.906021 - DATA
INFO:quic-client:2025-02-06 17:40:37.906056 - DATA
INFO:quic-client:2025-02-06 17:40:37.906085 - DATA
INFO:quic-client:2025-02-06 17:40:37.906109 - DATA
INFO:quic-client:2025-02-06 17:40:37.906134 - DATA
INFO:quic-client:2025-02-06 17:40:37.906155 - DATA
INFO:quic-client:2025-02-06 17:40:37.906177 - DATA
INFO:quic-client:2025-02-06 17:40:37.906198 - DATA
INFO:quic-client:2025-02-06 17:40:37.906219 - DATA
```

Figure 3: Example of QUIC packets captured using PyShark.

Results and Discussion
The aggregated results from the benchmarking tests and the packet-level analysis highlight:

- **TCP+TLS:**
  - The TCP+TLS captures confirm efficient segmentation and a proper sequence of TLS handshakes.
  - The QUIC captures validate the operation of multiplexed streams and consistent encryption (DATA frames), although the overall performance is lower.

5 Conclusions

The results of this study provide a detailed comparison between a traditional TCP+TLS file transfer solution and a modern QUIC-based implementation. Our experiments, conducted under loopback conditions using a 1GB file, showed that the TCP+TLS implementation exhibits extremely low handshake latency and round-trip times, which is expected in a loopback environment. When evaluating file transfers, this solution achieved an high throughput, with upload and download speeds reaching over 280 MB/s. In contrast, the QUIC implementation demonstrates higher handshake (0.019117 s) and RTT (average 0.005623 s) values. Although these numbers are still relatively low in absolute terms, they are still worse than those observed with TCP+TLS. The biggest difference was found in the throughput values, which in QUIC's case were in the low single-digit MB/s range (approximately 1.15 MB/s for both uploads and downloads). This contrast in overall performances indicates a substantial performance bottleneck in the current QUIC setup, possibly due to its immature implementation in Python and the additional overhead of handling multiplexed streams asynchronously.

References

[1] Jon Postel. *Rfc0793: Transmission control protocol*. 1981.

[2] Mark Allman, Vern Paxson, and Ethan Blanton. *TCP congestion control*. Tech. rep. 2009.

[3] Neal Cardwell et al. “Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time”. In: *Queue* 14.5 (2016), pp. 20–53.

[4] Adam Langley et al. “The quic transport protocol: Design and internet-scale deployment”. In: *Proceedings of the conference of the ACM special interest group on data communication*. 2017, pp. 183–196.