POLITECNICO DI TORINO

Master Degree course in Computer Engineering

Master Degree Thesis

A study of congestion control schemes on QUIC

Supervisors
Prof. Guido Marchetto
Prof. Alessio Sacco

Candidate
Alireza Ashtari

Academic Year 2022-2023

Acknowledgements

I would like to express my deepest gratitude to all those who have supported me throughout my academic journey, making this thesis possible.

First and foremost, I am profoundly thankful to my family, whose unwavering love, encouragement, and belief in my abilities have been my constant source of strength. To my parents, Gholamreza and Fatemeh, your sacrifices and guidance have been instrumental in shaping my academic pursuits. To my dear brother Amirhossein and sister Sana, your support and understanding have meant the world to me.

I extend my heartfelt appreciation to my professors, Prof. Guido Marchetto and Prof. Alessio Sacco, for their invaluable mentorship, expert guidance, and continuous support throughout this research. Your insights and dedication to the field have inspired me and enriched my academic experience.

I would like to thank all the teachers, colleagues, and friends who have shared their knowledge and insights with me along the way. Your contributions have been instrumental in shaping my understanding of the subject matter.

Last but not least, I want to express my appreciation to all the participants who contributed to this study. Your willingness to share your time and insights has been pivotal to the success of this research.

This journey has been challenging, yet incredibly rewarding, and I am deeply thankful to everyone who has played a part in it.

Abstract

This thesis aims to advance our understanding of congestion control in QUIC and demonstrate how reinforcement learning can be leveraged to enhance its performance, ultimately contributing to the improvement of network communication in various scenarios.
QUIC (Quick UDP Internet Connections) is a transport layer protocol developed by Google, designed to improve web browsing experience by reducing latency and enhancing reliability. Congestion control plays a crucial role in QUIC by dynamically adjusting the sending rate of data based on network conditions, ensuring optimal utilization of available bandwidth while avoiding network congestion.
In our investigation of the performance of the QUIC (Quick UDP Internet Connections) protocol in network environments, i focused particularly on its congestion control mechanisms. i compared QUIC to HTTP, seeking insights into whether QUIC consistently outperforms its predecessor. Surprisingly, our findings revealed that QUIC did not consistently provide superior performance. In fact, it sometimes exhibited poorer performance than HTTP.
Traditionally, congestion control algorithms have been handcrafted using predefined heuristics, which may not always adapt well to changing network dynamics or provide optimal performance across different scenarios.Recognizing the need for improvement, i embarked on a journey to fine-tune QUIC, with a particular emphasis on its congestion control parameters.
Our goal was to optimize both throughput and Round-Trip Time (RTT) to unlock the full potential of QUIC. To achieve this, i turned to Reinforcement Learning (RL), a cutting-edge technique known for its adaptability and ability to optimize complex systems. By utilizing RL, i aimed to create a model capable of dynamically adjusting congestion control parameters within QUIC.
By leveraging RL techniques, it is possible to train an intelligent agent to learn congestion control policies directly from raw feedback obtained during interactions with the network environment. The agent interacts with the QUIC stack and observes network metrics such as round-trip time, packet loss rate, and available bandwidth. It then takes actions by adjusting the sending rate of packets or changing parameters of congestion control algorithm accordingly.
During the training process, the RL agent explores different actions and learns which ones lead to better overall performance. By utilizing methods like deep Q-learning or policy gradient algorithms, it gradually improves its decision-making abilities over time. The agent's objective is usually defined as maximizing throughput while minimizing delay and packet loss.

The advantage of using RL for QUIC congestion control lies in its ability to adaptively respond to dynamic network conditions without relying on predefined rules or assumptions about network behavior. This allows for more efficient utilization of available bandwidth and improved user experience.
Furthermore, RL-based approaches can also handle complex scenarios that were difficult to account for using traditional congestion control algorithms.

Under varying network conditions, our experiments yielded promising results. Not only did QUIC's performance significantly improve in terms of throughput and Round Trip Time (RTT), but the network also remained stable. This stability is a crucial indicator of success, as it ensures a seamless user experience across diverse scenarios.
In conclusion, our research illuminates the dynamic nature of network performance and the challenges associated with optimizing modern transport protocols like QUIC. By harnessing the capabilities of RL, i have not only enhanced QUIC's throughput and RTT but also ensured the stability of the network. This work represents a substantial step towards realizing the full potential of QUIC in the ever-evolving landscape of internet communication.

Contents

Chapter 1

Introduction

The transmission control protocol (TCP) has served as the foundation of the internet ever since the first network nodes communicated in 1969. For over four decades, TCP's reliability, congestion control, and streamlined connections have enabled virtually all applications and services I use today, from web browsing and video streaming to cloud computing and online shopping. However, the internet landscape has evolved tremendously since TCP's inception, with new trends in mobility, bidirectionality, and interactivity exposing performance limitations in TCP's decades-old design.Applications ranging from online gaming and video conferencing to augmented/virtual reality and real-time collaboration demand levels of responsiveness and throughput that legacy TCP struggles to deliver.

QUIC (Quick UDP Internet Connections) represents a new generation of transport protocols designed specifically for today's internet landscape [8]. Originally developed at Google in 2012, QUIC aims to significantly improve connection establishment, congestion control, and overall performance for latency-sensitive traffic relative to TCP. QUIC runs atop UDP rather than TCP to reduce handshake latency, provides stream multiplexing over a single connection to avoid head-of-line blocking, and implements state-of-the-art congestion control algorithms tailored for high bandwidth and low delay. With rapid adoption at Google's servers and broad standardization efforts by the IETF, QUIC is well positioned to eventually supersede TCP as the default transport protocol powering the modern internet. However, realizing QUIC's full potential requires ongoing research and optimization, particularly around security, mobility, and congestion control.

This thesis specifically focuses on evaluating and improving QUIC congestion control relative to longstanding TCP algorithms.Congestion control constitutes the algorithms that dynamically throttle transmission rates to match available network capacity, avoiding queue buildup, packet loss, and instability. TCP congestion control evolved through the 1980s and 90s from early schemes like Tahoe, Reno, and NewReno to more modern innovations such as CUBIC, Compound, Vegas, and BBR. TCP CUBIC, first introduced in Linux kernel 2.6.19 in 2006, uses a cubic function to increase the congestion window

more aggressively compared to prior TCP variants, enabling higher throughput yet retaining Reno's stability and loss resilience. The default congestion control algorithms implemented in QUIC draw upon similar principles as CUBIC, probing available bandwidth during times of low congestion while backing off more rapidly during congestion events.

This thesis conducts a comprehensive evaluation comparing QUIC and TCP congestion control over both emulated network environments and live internet paths exhibiting diverse bottlenecks and variability. The analysis spans critical performance metrics including throughput, round-trip time, and packet loss resiliency. Under static network conditions, our results confirm QUIC's advantages for short flow completion time. However, TCP variants like CUBIC and BBR demonstrate higher throughput over long-lived flows and stronger stability in the face of sudden congestion spikes or fluctuating capacity. These findings highlight opportunities to further optimize QUIC congestion control, especially in volatile network environments.

To address these challenges, I develop a novel reinforcement learning (RL) agent to dynamically adapt QUIC congestion parameters based on real-time network state observations. Reinforcement learning is well-suited to sequential decision making under varying conditions. I model congestion control as a Markov decision process, where the agent continuously observes the environment state (e.g. recent throughput, RTT samples, and loss events) and selects actions to adjust CUBIC parameters accordingly. By extensively training this RL agent through simulated network environments exhibiting noise, delays, and capacity changes, it learns a policy mapping states to optimal actions that maximizes long-term reward. I define the reward function based on a combination of throughput, RTT, and connection stability. I validate this RL-optimized congestion control scheme against default QUIC, TCP CUBIC, and BBR over public internet paths prone to congestion and volatility.The results demonstrate the agent's ability to dynamically adapt its congestion window and pacing rate based on real-time network conditions to significantly improve throughput and latency compared to static schemes.

I further analyze the agent's learned policy to extract key insights into the correlations between observed states and corresponding actions. Our work represents the first demonstration of a reinforcement learning-based congestion control scheme practically deployed and evaluated over live network paths. The techniques developed in this thesis could generalize beyond QUIC to improve TCP performance and robustness as well.

In summary, this thesis provides a comprehensive benchmarking of QUIC congestion control mechanisms compared to widely used TCP variants over diverse network environments. Our evaluations reveal specific scenarios where further QUIC optimizations can unlock substantial performance gains. Leveraging cutting-edge reinforcement learning algorithms, I design a novel congestion control agent that can automatically adapt its sending rate based on real-time network state to improve throughput, latency, and resilience. Looking forward, reinforcement learning represents a promising approach to not only optimize QUIC but also address longstanding performance limitations in TCP itself.

This work helps pave the way towards a new generation of cognitive, self-tuning transport protocols ready to power the emerging landscape of interactive and delay-sensitive internet applications.

1.1 Organization of the Thesis

This thesis report is organized into Six main chapters starting from the Introductory chapter, followed by related works, underlying concepts and theoretical background, methodology, experiment, results and analysis, and finally, the conclusion and future works.

Chapter 1

The first chapter introduces the general background of this thesis and the motivation behind this research. Then, I will describe the Aim and Objectives of the research and explain the research questions and methods. Finally, limiting the scope of this research.

Chapter 2

The second chapter serves as the Background on Fundamental Technologies and Tools introduces the underlying concepts and theoretical background of QUIC protocol and its implementation. it explains about related works that have been done, background about QUIC and main motivation and it's mechanisms and specifically about congestion control and Reinforcement learning .

Chapter 3

The third chapter introduces the underlying concepts and Literature review of Reinforcement Learning and its implementation. It explains different part of RL and my contribution on developing RL's environment.

Chapter 4

The fourth chapter describes the Experimental and numerical Evaluation and the proposed test framework, providing an understanding of differential testing, the design structure of the framework, the experiment in detail, starting from the testing design, performance metrics, implementation, analysis.

Chapter 5

The fifth chapter presents the performance evaluation results, describing the results obtained and the data analysis regarding some aspects of performance.

Chapter 6

The sixth chapter concludes the result of this research and the recommendations for future research.

Chapter 2

Background on Fundamental Technologies and Tools

2.1 Background

The advent of the digital era has transformed technologies from optional tools into fundamental necessities across all aspects of life, including economies, societies, and personal lifestyles. Although existing technologies have matured, research and development continues to pursue further enhancements and opportunities.

Current emerging technologies predominantly involve rapid, massive data processing, including Artificial Intelligence, Edge Computing, the Internet of Things, and Virtual and Augmented Reality.

Reflecting technological proliferation, data generation and information flow on the internet have surged, especially during coronavirus pandemic in 2021. With global lock downs, digital technology became crucial for work, school, and socializing. During past years, internet growth and This massive data flow raises website performance concerns. Research indicates a 1 second to 3 second slowdown in page load times can decrease traffic by 32%, while a 10 second delay can plummet traffic by 123%.

This internet expansion risks network congestion and performance declines. It could also enable more cyberattacks on sensitive data transfers. However, HTTPS usage has steadily risen 300% as security awareness spreads. Although HTTP/2 over TCP and TLS enables congestion control and security, issues remain like TCP head-of-line blocking. TLS is not mandatory either. When enabled, TLS handshake increases round-trip times.

Seeking enhancements, protocols like SCTP, SPDY, and TCP Fast Open emerged. But SCTP saw minimal public adoption, SPDY was deprecated for HTTP/2, and TCP Fast Open suffered from ossification, with unchanged middleboxes dropping unfamiliar packets.

In 2012, Google engineer Jim Roskin developed QUIC to enable Quick UDP Internet

Connections. After public release in 2013 and broad experimentation, IETF standardization began in 2015. The QUIC working group collaborated with the HTTP group on HTTP over QUIC as HTTP/3. In 2021, QUIC was finalized as IETF RFC 9000, supported by RFC 8999, 9001, and 9002.

As an emerging technology, QUIC presents abundant research and experimentation opportunities, with the potential to solve persistent problems and replace TCP. Despite a long development process, QUIC already serves almost 8% of websites. It is also implemented in Google Chrome and Google services. While adoption is gradual, QUIC could prove one of history's most influential transport protocol breakthroughs.

2.2 QUIC Motivation

The evolving landscape of network interactions, spurred by the rapid expansion of the mobile Internet and the advent of the Internet of Things (IoT), has triggered heightened demands for enhanced network efficiency and greater responsiveness in web services.

Explosive Growth of Mobile Internet and IoT: The mobile Internet is experiencing rapid growth, and the proliferation of IoT devices is further amplifying this trend. This proliferation has ushered in a diverse array of approaches through which both devices and users engage with networks, culminating in a substantial surge in the volume of data traversing these networks.

Elevated User Expectations: Consequently, users now harbor heightened expectations for more efficient network transmission and swifter web service performance. Irrespective of the prevailing network conditions, users anticipate web pages and applications to provide prompt and dependable responses.

Navigating Challenges in Unpredictable Networks: Developers have been confronted with formidable challenges related to network stability, particularly in scenarios characterized by highly erratic network conditions. These conditions encompass recurrent transitions between distinct Wi-Fi networks, sporadic utilization of cellular data, and intermittent disruptions in cellular signal reception. Such fluctuations engender instability and unreliability in mobile Internet connectivity, leading to exasperatingly protracted loading times for web content.

Limitations of TCP: The extensively employed TCP protocol has encountered difficulties in enhancing performance within these capricious network environments. TCP's connection-oriented nature, coupled with its approach to congestion control, often results in delays and suboptimal performance, rendering it less suitable for networks with high variability.

UDP as an Alternative: In contrast, the User Datagram Protocol (UDP) offers

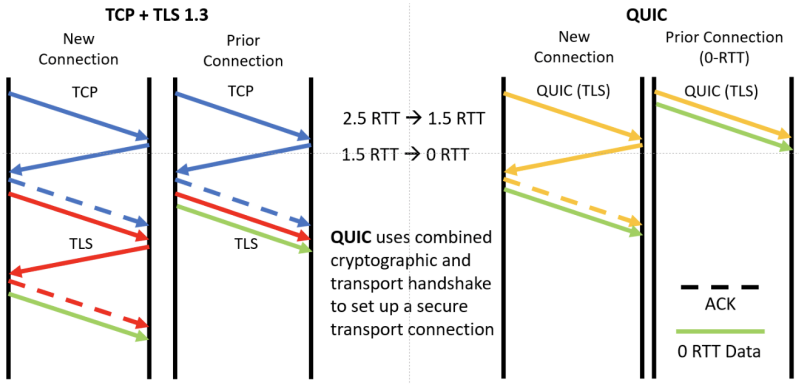
an alternative. It operates without the need for establishing connections and provides efficiency, rapid data transfer, and lower resource utilization. However, it lacks some of the reliability mechanisms inherent in TCP.

Google's Innovative Response - QUIC: Recognizing these challenges, Google has proactively tackled them by devising a groundbreaking protocol known as QUIC [8]. Beyond the capabilities of UDP, QUIC introduces bidirectional bandwidth control, thereby enabling proficient management of network congestion. This pioneering development seeks to address the intricacies posed by unpredictable network conditions, ultimately elevating the overall user experience by bolstering the speed and reliability of web services and applications.

QUIC emerged from Google as an innovation aimed at enhancing the functionality of TCP (Transmission Control Protocol) and TLS (Transport Layer Security) in the context of internet applications and web navigation. Noteworthy among the primary drivers and enhancements associated with QUIC in comparison to the amalgamation of TCP and TLS are as follows:

Reduced latency in connection establishment - QUIC streamlines the cryptographic and transport handshake process, condensing it into a single round trip, as opposed to the 1-3 round trips required by TCP+TLS. This enhancement contributes to improved webpage loading times.Figure 2.1, 2.2

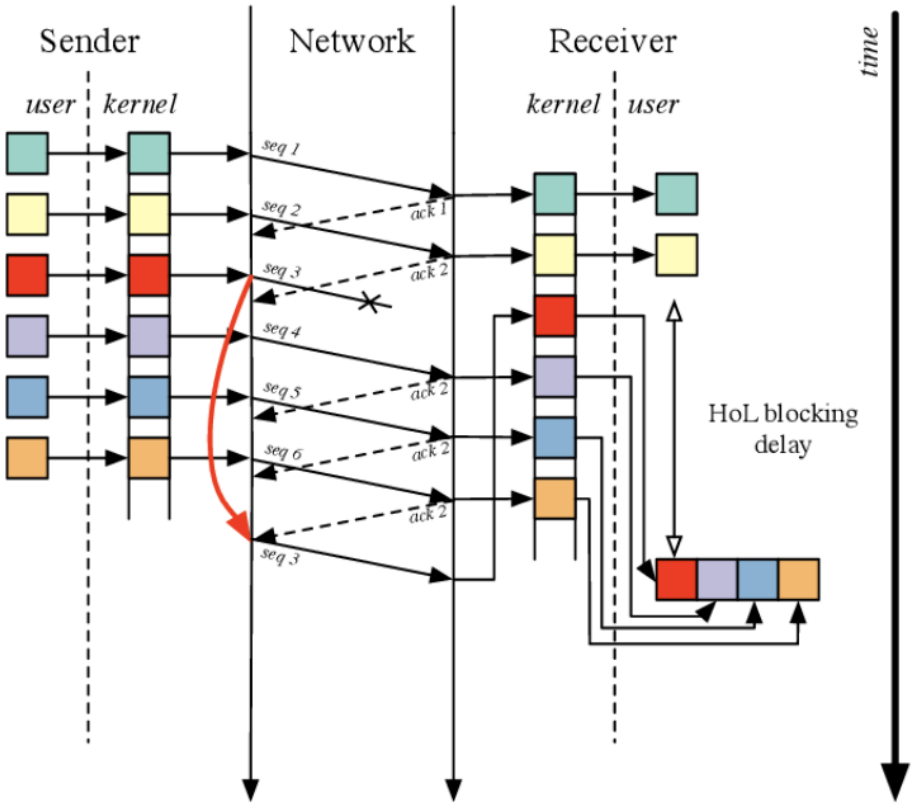
Figure 2.1. TCP vs QUIC connection setup



Enhanced congestion control - QUIC incorporates its own congestion control algorithms, surpassing the performance of TCP's slow start mechanism and avoiding issues related to head-of-line blocking, which are prevalent in TCP.

Non-blocking multiplexing - QUIC streams enable the simultaneous handling of requests and responses over a single connection without encountering blocking, a feature

Figure 2.2. multiple segment are delayed due to (b) TCP lost packet break the chain con- packet loss resulting HoL Blocking



that sets it apart from TCP.Figure 2.3 , 2.4

Network address-agnostic connections - QUIC connections are capable of adapting to changes in network addresses, ensuring uninterrupted communication even in the face of network address alterations.

Forward error correction - QUIC offers forward error correction for all transmitted data, enabling the receiver to rectify data losses.

Unlike protocols implemented in the kernel space, QUIC operates at the application layer in userspace. This allows rapid iteration for QUIC development and modifications without requiring operating system kernel updates on clients and servers.

The userspace QUIC stack enables more agile evolution compared to TCP protocols. As illustrated in the figure, the QUIC stack resides above the kernel, separate from lower-level networking layers.Figure 2.5

Figure 2.3. TCP Head of Line Blocking

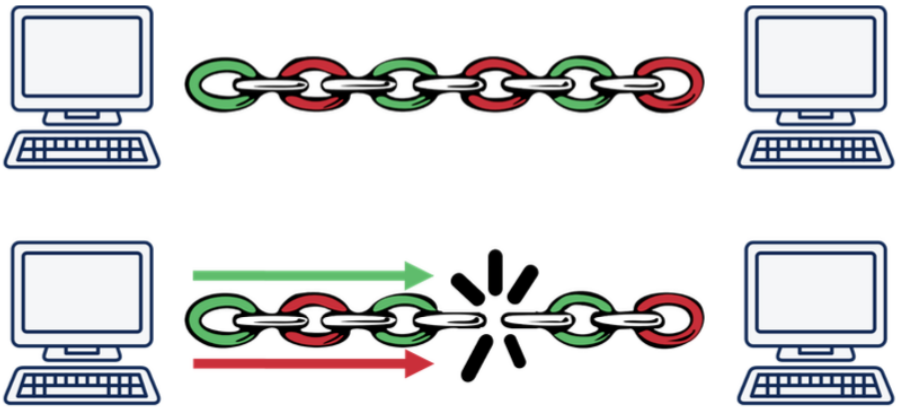
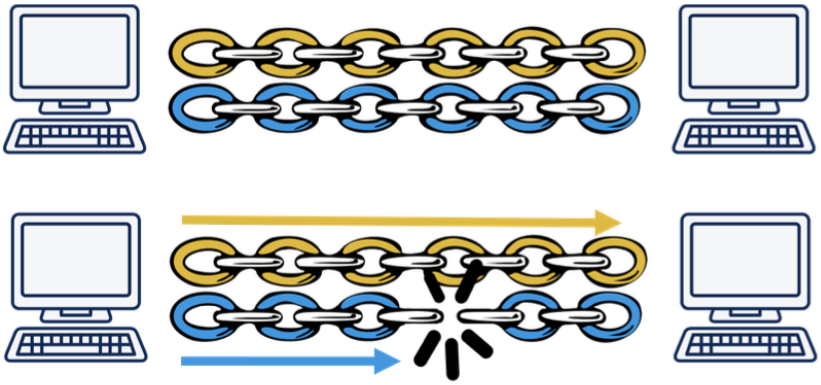


Figure 2.4. QUIC lost packet only affect specific stream

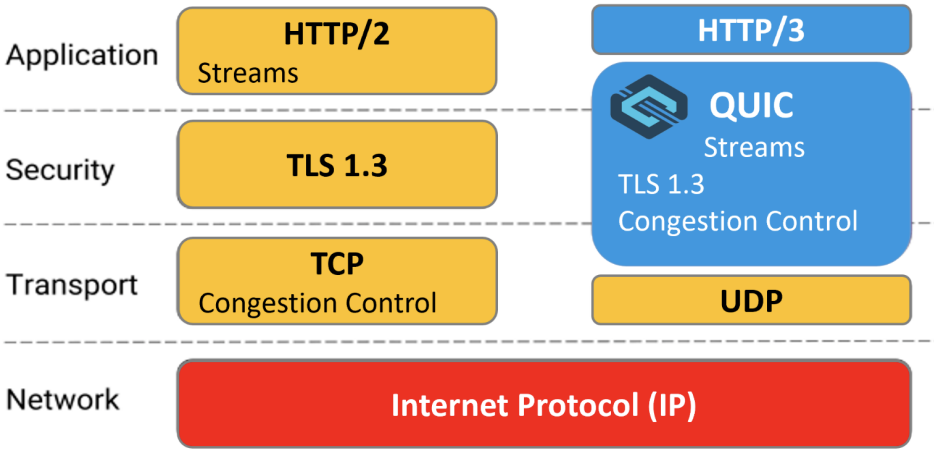


2.3 QUIC Mechanisms

2.3.1 Connection establishment

QUIC's connection establishment process is designed to be faster and more efficient. QUIC swiftly initiates connections through a cryptographic handshake, enabling mutual identity verification and the agreement on encryption keys for data security. This handshake is accomplished with just 1 or 2 round trip times (RTTs), in contrast to the 3-way handshake mandated by TCP. I can describe the QUIC connection establishment process step by step [8].

Figure 2.5. QUIC stack compared to TCP



Step 1: Initial Packet Exchange

The QUIC client sends a QUIC Initial packet (often referred to as a "CHLO" or Client Hello) to the server. The CHLO packet includes information like supported QUIC versions, cryptographic parameters, and other necessary details.

Step 2: Server Response

The QUIC server receives the CHLO packet and processes it. The server responds with a QUIC Initial packet (often referred to as a "SHLO" or Server Hello). The SHLO packet includes the server's chosen parameters, such as cryptographic keys, supported QUIC version, and other relevant information.

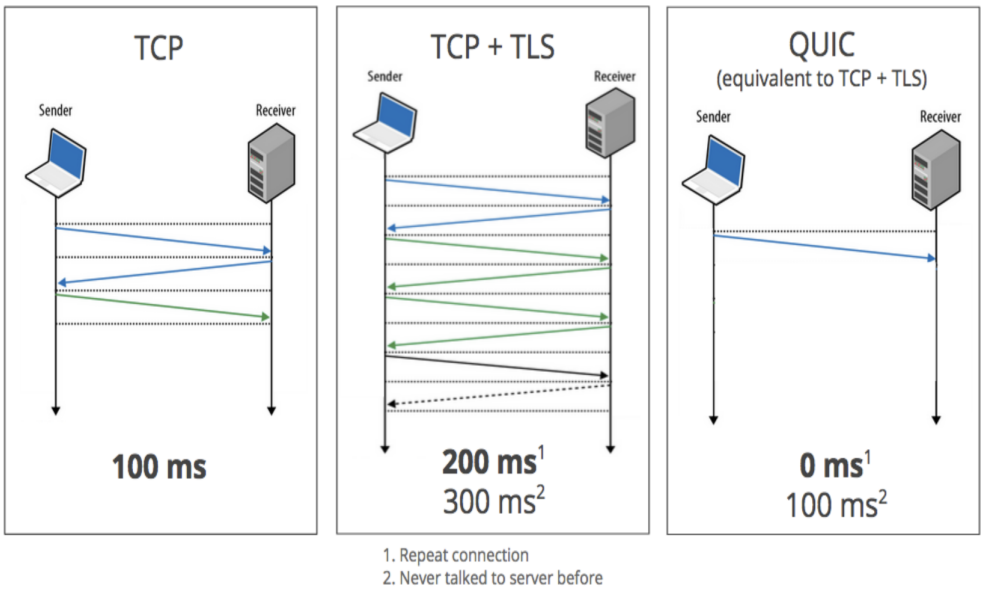
Step 3: Cryptographic Handshake

Both the client and server perform a cryptographic handshake to establish secure communication. They exchange cryptographic parameters and keys to set up encryption for the connection. Once this handshake is complete, the client and server can securely exchange data.

Step 4: Data Exchange

With the cryptographic handshake complete, the QUIC connection is established. The client and server can now exchange data packets over the QUIC connection. Data packets can include HTTP requests, responses, or any other application-level data.

Figure 2.6. Zero RTT Connection Establishment [1]



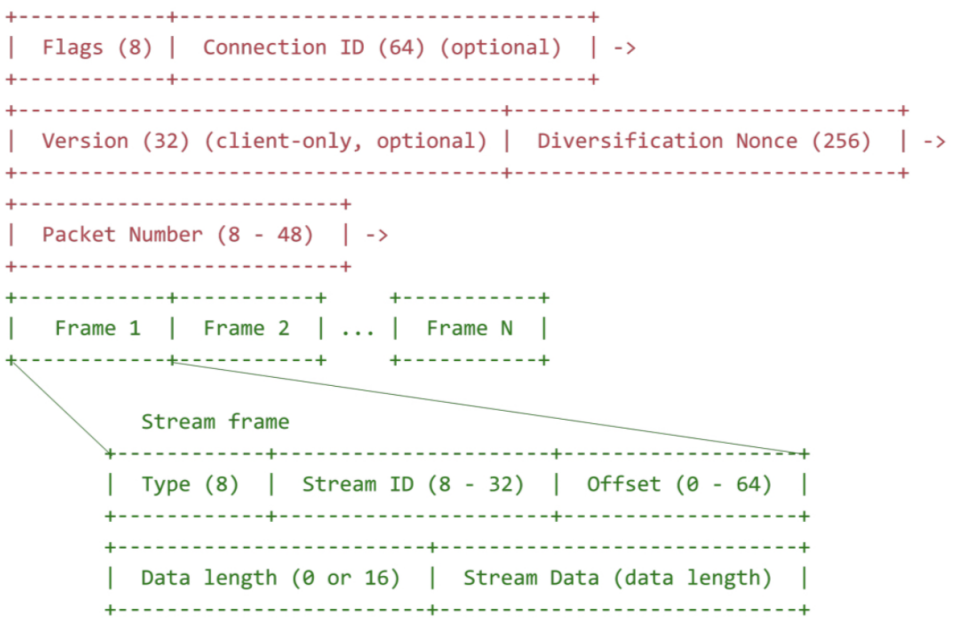
2.3.2 Multiplexing

QUIC facilitates the concurrent transmission of diverse data streams over a solitary connection. Each individual stream operates independently, ensuring that if one encounters issues such as packet loss or delays, it does not impede the progress of other streams. This characteristic effectively mitigates the head-of-line blocking challenge typically associated with TCP. [8] [10].
In the context of QUIC, multiplexing refers to the ability to transmit multiple data streams simultaneously within a single QUIC connection.
Each data stream is assigned a unique identifier, known as a "Stream ID," which allows both the sender and receiver to distinguish between different streams. Multiplexing enables various types of data, such as HTTP requests, responses, and other application-level data, to be sent concurrently over the same connection.
Since each stream operates independently, issues affecting one stream, such as packet loss or latency, do not impact the progress of other streams. This is in contrast to TCP, where head-of-line blocking can occur, causing delays in the delivery of data from multiple streams when one encounters issues.
The non-blocking nature of QUIC multiplexing contributes to improved overall performance and responsiveness, particularly in scenarios where multiple resources need to be loaded simultaneously, such as modern web pages with numerous assets.

2.3.3 Packet number Encryption

QUIC employs encryption to protect a significant portion of its transport layer header, including packet numbers. This safeguard shields against the monitoring of traffic patterns and sequence numbers by network devices, significantly bolstering both privacy and security in comparison to the unencrypted TCP header. [8] [10]

Figure 2.7. QUIC packets are fully authenticated

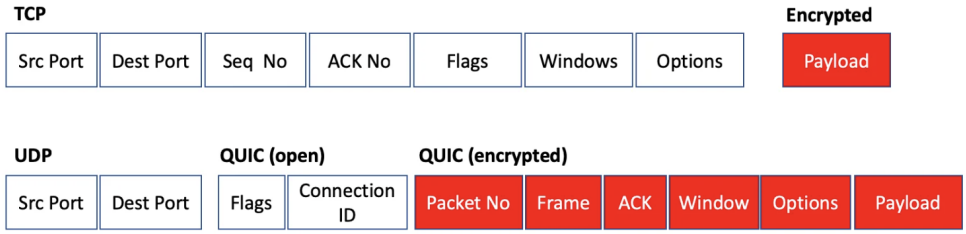


The packet number plays a crucial role in each QUIC packet's header, serving to identify packet loss and maintain sequencing integrity. In TCP, packet sequence numbers are transmitted without encryption, rendering them susceptible to passive observation. However, in QUIC, packet numbers are automatically encrypted before transmission using an AEAD cipher (Authenticated Encryption with Associated Data). These encryption keys are established during the initial cryptographic handshake between the client and server.

Only entities possessing the correct decryption keys, namely the client and server, can decipher the packet numbers and employ them for loss detection and sequence maintenance. This encryption process significantly obstructs network devices, such as routers or firewalls, from deducing information regarding traffic patterns or sequences by inspecting unencrypted headers.

By encrypting packet numbers, QUIC effectively thwarts overt passive monitoring of packet flows across the network path, reinforcing privacy and impeding surveillance efforts. Furthermore, the encryption strategy employed for packet numbers is designed

Figure 2.8. QUIC packets Encryption vs TCP [2]



to withstand replay attacks. Each key is employed for a brief duration before being discarded, making it possible to detect and reject replayed packets that rely on outdated keys.

It's worth noting that there are limited situations in which the packet number may be transmitted without encryption, such as during stateless retries in the connection setup process.

2.3.4 Connection migration

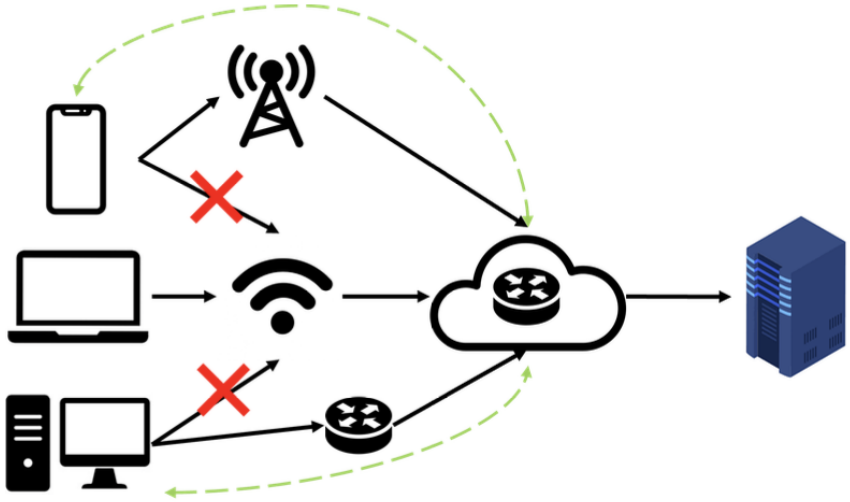
QUIC connections employ a Connection ID rather than the conventional 4-tuple consisting of source/destination IP addresses and ports. This design choice grants the ability to shift the connection to an alternative network path or endpoint address. Importantly, the Connection ID is intentionally dissociated from network-layer identifiers such as IP addresses Figure 2.9. [8] [10]

To facilitate connection migration, the new network path is probed for connectivity. Once deemed functional, the Connection ID comes into play, allowing the QUIC session to seamlessly continue on the fresh path. For instance, a smartphone can transition from a WiFi network to LTE, and the QUIC connection smoothly transitions to the new interface. This endpoint migration ensures uninterrupted service even if the server's IP address changes, preserving higher-level connections.

QUIC adeptly manages packet loss during migration through acknowledgment and retransmission mechanisms, while session state is reestablished on the new path. Migration can be instigated by either clients or servers and may utilize techniques like DHCP for network change detection. Notably, this process occurs without necessitating any application layer awareness or specific triggers, affording mobility and deployment adaptability that TCP, tethered to a fixed 4-tuple address, lacks.

During migration, both the original and new network paths briefly coexist. QUIC

Figure 2.9. QUIC Connection Migration



sends probe packets over the new path to assess connectivity. Once validated, the connection fully transitions to the new path, leaving the old path behind. For minor migration delays (less than 200ms), QUIC can buffer application data during the handover to prevent disruptions in data delivery. However, longer delays may entail a brief pause in data transfer during the handover.

Furthermore, migration can occur repeatedly, allowing connections to traverse different paths as clients move. Servers employ the Connection ID to retrieve connection state and seamlessly continue on the new path. Clients can initiate migration, while servers can also enforce it in certain scenarios. NAT rebinding support aids in handling IP address changes during migration.

Notably, connection migration remains fully encrypted end-to-end between the communicating endpoints. The new path may offer distinct properties, such as reduced latency or enhanced bandwidth, immediately enhancing the connection's performance post-migration.

Overall, connection migration fortifies QUIC against network alterations and mobility, enhancing the user experience during transitions across diverse networks and endpoints.

2.3.5 Forward Error Correction (FEC)

QUIC employs Forward Error Correction (FEC) techniques, including parity check codes like Reed-Solomon codes. FEC data enables the recovery of lost packets without the

need to wait for retransmissions. To achieve this, QUIC proactively transmits FEC data alongside regular stream data in separate redundant packets. [8] [7]

The FEC data serves as a means to reconstruct the original payload in case a limited number of packets within a block are lost. Consequently, the receiver can promptly recover lost packets and reassemble out-of-sequence packets, all without initiating retransmission requests. This mechanism significantly reduces the delay in recovering from packet loss when compared to the retransmission process in TCP.

The effectiveness of QUIC FEC is most pronounced in environments where packet loss rates are moderate, typically ranging from 1% to 5%. In scenarios characterized by either very low or exceptionally high loss rates, the benefits of FEC become less significant.

One notable advantage of QUIC is its ability to adapt the strength of FEC dynamically based on observed network conditions and loss rates. However, it's important to note that sending redundant FEC data does introduce overhead. Therefore, there exists a tradeoff between loss resilience and the incurred overhead.

It's essential to highlight that QUIC FEC exclusively addresses data loss issues; it does not serve as a means to validate data integrity. Authentication remains a necessary component to detect any tampering attempts.

In summary, FEC empowers QUIC to shield the application layer from the effects of packet loss, enhancing the perceived data delivery performance, especially in loss-prone network environments.

2.3.6 Flow Control

QUIC incorporates flow control mechanisms at both the connection and stream levels to manage data transmission effectively. [8]

Connection-level flow control is designed to restrict the total data volume transmitted over the QUIC connection, thus averting the risk of resource depletion. Each QUIC endpoint communicates a maximum connection flow control limit to its peer, typically a few megabytes. As data is transmitted, it depletes this connection limit. Additional data can only be sent when the peer grants an increase in this limit.

Stream-level flow control, on the other hand, applies to each individual stream and aims to prevent any single stream from monopolizing connection resources. Each stream is assigned its own maximum stream data limit, typically set at 64 kilobytes per stream. Data transmitted on a stream reduces the available capacity according to that specific stream's limit. Crucially, if a stream becomes blocked, it does not impede the progress of other streams.

The communication of flow control limits is achieved through MAX_DATA and MAX_STREAM_DATA frames within QUIC packets. In the event that received data surpasses a flow control limit, the endpoint discards the excess data and may even terminate the connection.

QUIC's flow control framework serves two vital functions. Firstly, it exerts back pressure on senders, ensuring that they do not overwhelm the receiver or the network with excessive data. Secondly, it fortifies the protocol against resource exhaustion attacks, enhancing security in the process.

2.4 HTTP/3

HTTP/3 is a new iteration of the Hypertext Transport Protocol designed to fully leverage the capabilities of the QUIC protocol. It delivers new solutions for HTTP over QUIC, providing beneficial HTTP features like stream multiplexing, per-stream flow control, and low latency connections (Krasic et al., 2022). Modifications were necessitated by QUIC's differing nature from TCP, while some HTTP/2 features are subsumed, allowing HTTP/3 to delegate particular tasks to QUIC for resolving TCP head-of-line blocking issues. HTTP/3 utilizes similar semantics and internal framing as HTTP/2.

Communication within each stream uses frames, including a dedicated control stream conveying frames applying to the whole connection. HTTP/3 also adopts HTTP/2's server push mode and header compression, replacing HPACK with QPACK. Major divergences from HTTP/2 include faster handshakes enabling enhanced early data support, mandatory security, increased stream allowances, removed priority signaling, flow control for all frames and payload, distinct settings parameters, and error codes.Figure 2.10 [5]

2.5 Related work

2.5.1 QUIC Protocol VS TCP

Analyzing the performance of QUIC has posed a significant challenge for researchers over the years due to its lack of standardization and its continuous development and modifications. Additionally, the diverse design approaches, development processes, and disparities in features among implementations necessitate careful consideration when configuring and adjusting settings to conduct equitable performance assessments across various QUIC implementations. Nevertheless, even though QUIC has not reached full maturity, it has demonstrated the potential to outperform TCP connection times in specific scenarios.

QUIC does offer advantages over TCP in terms of reduced latency, adaptability, and simplicity at the application layer. However, its performance is significantly influenced by

Figure 2.10. Comparison between HTTP/2 and HTTP/3

Features	HTTP/2	HTTP/3
Transport	TCP	QUIC
Stream	HTTP/2	QUIC
Clear-Text version	Yes	No
Independent Streams	No	Yes
Header Compression	HPACK	QPACK
Server Push	Yes	Yes
Early Data	In Theory	Yes

variations in developer design choices and operator configurations, resulting in inconsistencies in performance outcomes across different implementations and testing scenarios. Notably, empirical performance tests conducted on production endpoints reveal that the mere deployment and use of QUIC do not automatically guarantee enhanced network and application performance in many practical use cases. [20]

In another investigation, it was discovered that QUIC demonstrates strong performance when dealing with brief interactions, like downloading small files or browsing websites. However, its throughput diminishes during lengthy sessions, such as large file downloads and video streaming. Nonetheless, QUIC's advantage lies in its ability to deliver lower latency, thereby enhancing video quality during video transmission and reducing the time required for initiating web workloads in comparison to TLS 1.3 over TCP. [14]

2.6 Congestion control

One of the most crucial components for enabling both fair and high utilization of Internet networks shared by numerous flows is transport-layer congestion control. The network's performance will start to suffer at some point if the demand for a resource exceeds the amount of that resource that is actually available. Congestion is the term for this circumstance. [6]
Congestion control's goal is to carry out the necessary processing in the event of system overload, ensuring that the system operates steadily and returns to its normal load level. Controlling traffic is an international effort. However, congestion does not affect UDP itself. Once in unrestricted usage, it will eat up the bandwidth of other network protocols that are "rule-worth" something.

In-Built Congestion Control: QUIC incorporates its congestion control mechanisms directly into the protocol, rather than relying on the operating system's TCP stack. This enables QUIC to be more responsive and adaptable to network conditions.

End-to-End Congestion Control: Like TCP, QUIC uses end-to-end congestion control, meaning that it relies on feedback from the receiver (acknowledgments) to gauge network conditions and adjust its sending rate accordingly.

2.6.1 Packet Number rising monotonically

To verify that the message arrived in good order, QUIC does not employ the byte order number and ACK features of TCP. Packet Number is employed by QUIC. If Packet N is lost, the Packet Number that retransmits Packet N is not N, but a number higher than N since each Packet Number is rigidly incremented. This makes resolving the ambiguity issue in TCP retransmission simple.

2.6.2 Calculate RTT time with accuracy

The QUIC ACK packet additionally includes the time between when the packet was received and when the reply ACK was sent. The incremental packet number may be utilized to determine the RTT with accuracy in this fashion.

2.6.3 QUIC Congestion Control Algorithms

CUBIC: QUIC's default congestion control algorithm is CUBIC, which is an evolution of the TCP CUBIC algorithm. CUBIC is designed to provide efficient congestion control with improved throughput and fairness characteristics. It uses a cubic function to calculate the congestion window size and is particularly effective in high-speed, high-bandwidth networks.

Pacing: QUIC includes built-in packet pacing, which helps smooth the transmission rate and reduce burstiness in the network. Pacing can help avoid congestion caused by sudden bursts of traffic.

RTT-Based: QUIC's congestion control algorithms often rely on round-trip time (RTT) measurements to estimate network congestion. Shorter RTTs indicate less congestion, allowing QUIC to increase its sending rate, while longer RTTs may indicate congestion, leading to a reduction in the sending rate.

Packet Loss Detection: QUIC uses packet loss as a signal of network congestion, similar to TCP. When packet loss is detected, QUIC reduces its congestion window size to prevent further congestion.

Bandwidth Estimation: QUIC estimates available network bandwidth by monitoring the rate at which acknowledgments are received and the round-trip time. This helps it adapt to changing network conditions.

Stream-Level Congestion Control: QUIC supports multiple streams within a single connection, and its congestion control operates independently for each stream. This allows for better fairness and adaptability, especially in scenarios where different applications have varying bandwidth requirements.

2.6.4 Window growth function for the CUBIC

The window growth function for the CUBIC congestion control algorithm is:

$$Wcubic = C(t - K)^3 + Wmax \tag{2.1}$$

Wcubic = CUBIC congestion window size
C = Scaling factor
t = Time since last window reduction
K = Time period for Wcubic to increase to Wmax after a window reduction
Wmax = Window size just before last reduction.

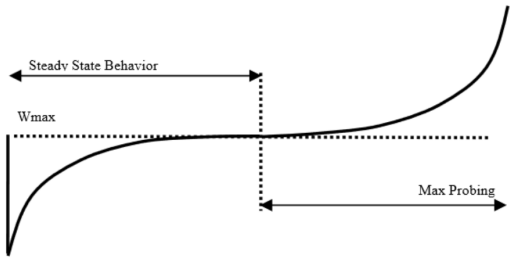
And K is calculated as:

$$K = \sqrt[3]{(Wmax * \beta/C)} \tag{2.2}$$

β = window reduction factor.

The cubic function causes the window to grow very fast initially after a reduction, then slower as it approaches Wmax. This aims to improve scalability while maintaining stability.. The C scaling factor controls the slope of the cubic curve. Higher C causes more aggressive growth. The β is reduction factor controls how much the window is decreased on loss. Higher β causes slower convergence. inproceedings

Figure 2.11. The Window Growth Function of CUBIC [18]

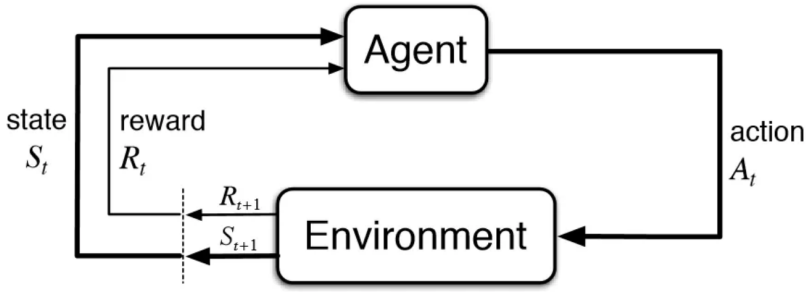


The growth function of CUBIC is seen in Fig. 2.11 with its origin at Wmax.After a window reduction, the window expands quickly, but as it approaches Wmax, it grows more slowly. The window increment virtually disappears at Wmax.Above that, CUBIC begins searching for additional bandwidth, and as it gets farther from Wmax, the window's development picks up speed. While the rapid expansion away from Wmax assures the protocol's scalability, the gradual growth around Wmax improves the stability of the protocol and raises network usage.

2.7 Overview of Reinforcement Learning

Reinforcement learning (RL) is a machine learning paradigm wherein an agent acquires the ability to sequence decisions by interacting with an environment. The primary objective of the agent is to maximize its cumulative reward over time by making choices that result in favorable outcomes. RL represents a subfield within the realm of artificial intelligence (AI) and has garnered substantial attention for its practical applications across diverse domains such as robotics, gaming, autonomous systems, and more. [16] [19] [4] [9]

Figure 2.12. Reinforcement Learning Cycle [13]



Here, I present a concise overview of the foundational concepts integral to reinforcement learning:

Agent: This refers to the entity responsible for learning and decision-making through interactions with the environment.

Environment: The external context in which the agent operates, where actions are executed, and from which feedback is received in the form of rewards.

State: A representation of the current configuration or condition of the environment, serving as the basis for the agent's decision-making process.

Action: The set of available choices or decisions that the agent can make at each given state, enabling it to interact with the environment.

Policy: The strategy or mapping that defines how the agent should behave, specifying the association between states and actions. The ultimate goal is to learn an optimal policy that maximizes long-term rewards.

Reward: A numeric signal, provided by the environment following each action taken, denoting immediate benefits or costs. The agent's overarching objective is to maximize the cumulative reward accumulated over time.

Value Function: A functional representation that estimates the anticipated cumulative reward, or value, of adhering to a specific policy in a particular state.

Q-Learning: A widely recognized RL algorithm, Q-Learning estimates the anticipated cumulative reward for selecting a particular action in a given state (Q-value). These estimations are subsequently used to refine the policy.

Exploration vs. Exploitation: A fundamental dilemma in RL that involves striking a balance between experimenting with new actions (exploration) and favoring actions perceived as optimal (exploitation).

Markov Decision Process (MDP): A mathematical framework that formalizes the core principles of RL, encompassing states, actions, rewards, and state transitions.

Learning Algorithms: RL employs learning algorithms to update the agent's policy or value estimates based on its interactions with the environment. Common RL algorithms include Q-learning, Deep Q-Networks (DQN), Policy Gradient methods, and Actor-Critic architectures.

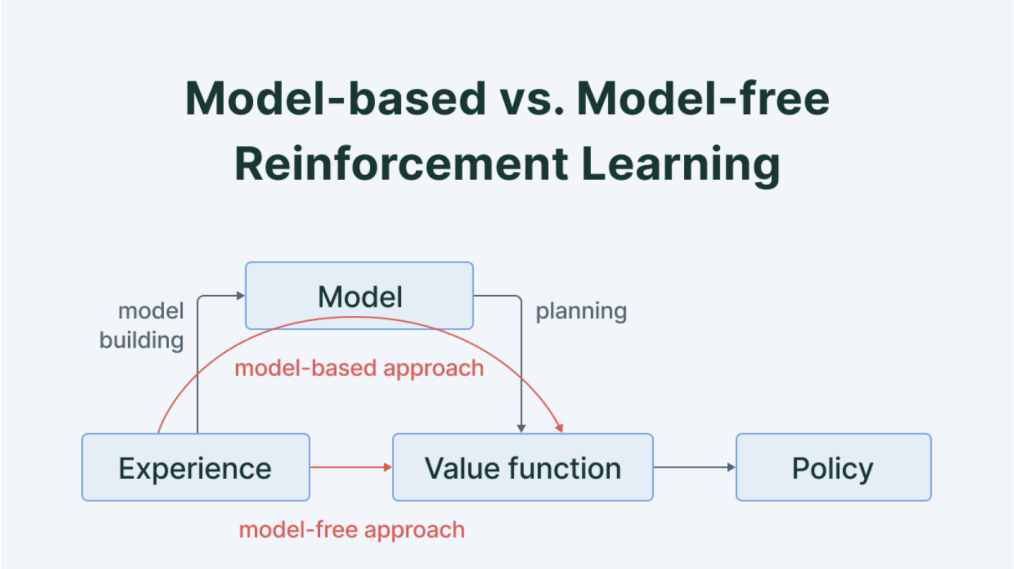
Policy Optimization: Policy optimization methods focus on directly improving the agent's policy to maximize expected rewards. These methods include Proximal Policy Optimization (PPO), Trust Region Policy Optimization (TRPO), and more.

Temporal Difference (TD) Error: TD error is a measure of the discrepancy between the predicted and actual rewards obtained by the agent. It is used in various RL algorithms to update value functions and policies.

Reinforcement learning has witnessed significant advancements, with the emergence of algorithms such as Deep Q-Networks (DQN), Proximal Policy Optimization (PPO), and various policy gradient techniques. These algorithms harness the capabilities of neural networks to effectively handle high-dimensional state spaces, rendering RL well-suited for tackling intricate tasks such as playing video games, orchestrating robotic systems, and optimizing resource allocation.

2.7.1 Model-based vs Model-free

Figure 2.13. Model-based vs Model-free [3]



There are two main types of Reinforcement Learning algorithms: Figure 2.13

1. Model-based algorithms
2. Model-free algorithms

Model-based algorithms

Model-based algorithms utilize the transition and reward functions to estimate the optimal policy.

- They are applicable when the environment is fully observable, allowing complete knowledge of how it reacts to various actions.
- The agent can access the environment model, including actions to transition between states, associated probabilities, and resulting rewards.
- This enables planning by thinking ahead. For static environments, model-based reinforcement learning is more appropriate.

Model-free algorithms

In contrast, model-free algorithms find the optimal policy with minimal knowledge of environment dynamics. They lack transition and reward functions to judge the best

policy.

- model-free algorithms find the optimal policy with minimal knowledge of environment dynamics. They lack transition and reward functions to judge the best policy.
- Model-free reinforcement learning is preferable when environmental information is incomplete.
- Real-world environments are often dynamic rather than fixed. For instance, self-driving cars operate in changing traffic conditions and diversions. In such scenarios with incomplete environment models, model-free algorithms tend to outperform other techniques.

2.7.2 Algorithms Of RL

RL includes multiple algorithms in order to train model which some of them are as follow:

Q-Learning [19]:

Q-Learning integrates policy and value functions to jointly assess actions based on their utility for obtaining future rewards. Quality values Q(s,a) are assigned to state-action pairs according to the expected future value given the current state and the agent's optimal attainable policy. After learning the Q-function, the agent identifies the highest quality action at a particular state s.

Once the optimal Q-function Q* is determined, the optimal policy can be derived by applying a reinforcement learning algorithm to find the action maximizing the value for each state. In essence, Q-Learning combines policy and value estimation to judge actions by their potential for maximizing cumulative future rewards based on the learned Q-function mapping states and actions to expected quality.

Figure 2.14. Q-Learning Formula [3]

$$q_*(s,a) = \max_{\pi} q_\pi(s,a)$$

In other words, Q* gives the largest expected return achievable by any policy π for each possible state-action pair.Figure 2.15

In the basic Q-Learning approach, we need to maintain a look-up table called q-map for each state-action pair and the corresponding value associated with it.Figure 2.16

SARSA [12]:

Figure 2.15. Q-Learning Cycle - Q-table [3]

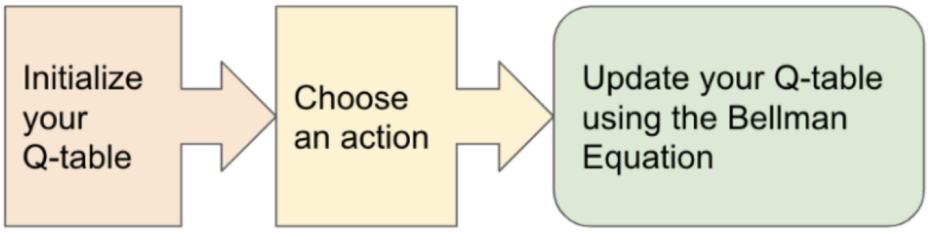
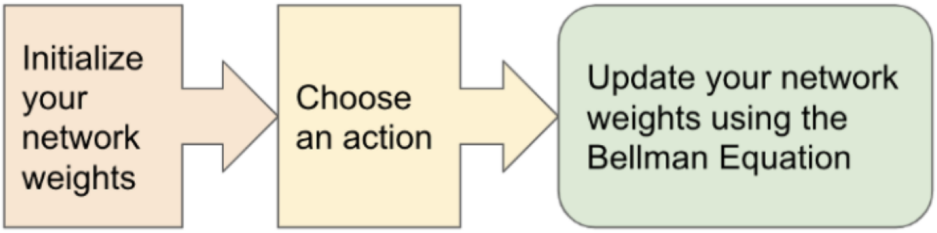


Figure 2.16. Q-Learning Formula - Bellman Equation [3]



On-policy TD control algorithm, updates Q(s,a) for the current policy. Uses experienced transitions (s,a,r,s',a') to update Q values. Policy improvement applies once Q values are accurate.

Deep Q-Networks [11]:
Uses deep neural nets to represent Q(s,a) with high dimensionality. Experience replay buffer breaks correlations and stabilizes training. Achieved human-level performance on Atari games.

Policy Gradients [17]:
Directly adjust policy parameters and Reduce variance using baselines and actor-critic approaches. Enable policies with stochasticity, good for continuous actions.

Markov Decision Process (MDP) [15]
Markov decision processes (MDPs) provide a formalism for sequential decision making, which underlies problems addressed by reinforcement learning. In an MDP, an agent called the decision maker interacts with an environment. These interactions occur sequentially over time.

At each timestep, the agent receives a representation of the environment state. Based on this state, the agent selects an action. The environment then transitions to a new

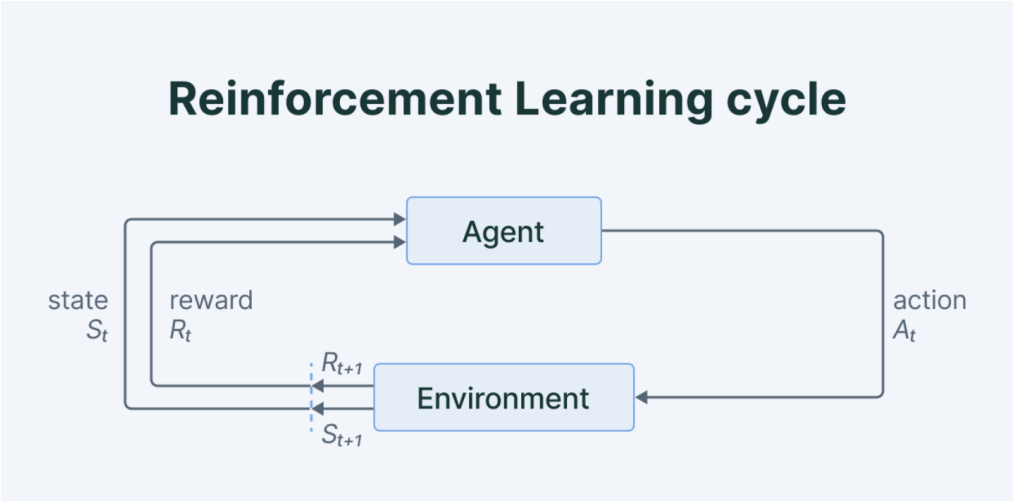
state, and the agent receives a reward as a consequence of its preceding action.

This cycle of state observation, action selection, state transition, and reward reception recurs sequentially, creating a trajectory capturing the sequence of states, actions, and rewards.

The reinforcement learning agent's objective is to maximize cumulative rewards received over the whole trajectory by taking optimal actions in each state. Rather than solely maximizing immediate rewards, the agent seeks to optimize long-term reward acquisition.

This iterative interaction between agent and environment, aiming to maximize rewards by selecting actions based on environment states, epitomizes the reinforcement learning approach known as Markov decision processes. The concept is clearly depicted in the provided. Figure 2.17

Figure 2.17. Q-Learning Formula [3]



2.7.3 Flow Of RL Process In Depth

Certainly, let's delve deeper into the flow of the Reinforcement Learning (RL) process (Figure 2.5), focusing on its key components and mechanisms:

Initialization:

At the start, the agent initializes its policy, which defines how it chooses actions based on states. This policy could be a neural network, a Q-table, or another function approximation method. In Deep RL, neural networks are commonly used to approximate the

policy (policy-based methods) or value functions (value-based methods).

Observation:

The agent perceives the current state of the environment. This state could be represented as raw sensor data, a set of features, or images, depending on the problem.

Action Selection:

The agent decides which action to take based on the observed state. This decision is guided by its current policy. Exploration vs. Exploitation: RL agents face the exploration-exploitation trade-off. They must balance trying new actions (exploration) to discover optimal strategies with choosing actions they believe are the best based on their current knowledge (exploitation).

Action Execution:

The agent takes the selected action, which affects the environment. This action might involve moving in a game, sending a control signal to a robot, or any other relevant action in the context of the problem.

Transition and Reward:

As a consequence of taking the action, the environment transitions to a new state. The agent receives a numerical reward signal from the environment, which quantifies the immediate benefit or cost of the action. The transition includes the current state, action taken, reward received, and the new state.

Experience Replay (for Deep RL):

To improve sample efficiency and stabilize learning, Deep RL agents often use experience replay. They store past experiences (state, action, reward, next state) in a replay buffer. During training, the agent samples mini-batches from the replay buffer rather than learning from sequential experiences.

Value Estimation (for Value-Based Methods):

In value-based RL, the agent estimates the value of states or state-action pairs. This estimation is based on the cumulative expected future rewards. Value functions like the Q-function (Q-learning) or the state-value function (V-learning) help the agent assess the desirability of different states or actions.

Policy Improvement (for Policy-Based Methods):

In policy-based RL, the agent directly updates its policy to maximize expected cumulative rewards. It employs gradient ascent to improve the policy, making actions that lead to higher rewards more likely.

Loss Calculation (for Deep RL):

In Deep RL, the agent calculates a loss function based on the predicted values (Q-values or policy) and target values. Common loss functions include Mean Squared Error (MSE) for Q-learning and policy gradient loss for policy-based methods.

Backpropagation and Optimization (for Deep RL):
The agent backpropagates the calculated loss through the neural network, adjusting its weights to minimize the loss. Optimization algorithms like Stochastic Gradient Descent (SGD), Adam, or RMSprop are often used for this purpose.

Target Networks (for Deep RL):

In value-based Deep RL (e.g., DQN), two neural networks are used: the online network and the target network. The target network provides stable target Q-values, reducing the risk of value estimate oscillations during training.

Temporal Difference Learning:

RL agents employ Temporal Difference (TD) learning to estimate the difference between predicted and actual rewards. This difference guides policy and value updates.

Policy Evaluation and Improvement Iteration:

Steps 2 to 12 are performed iteratively over multiple episodes or time steps. The agent continually learns from its experiences and refines its policy.

Convergence and Exploration:

The RL agent continues learning until it converges to an optimal policy, meaning it has found a policy that maximizes expected cumulative rewards. Exploration strategies like epsilon-greedy or others are gradually reduced over time as the agent becomes more confident in its policy.

Termination:

The training process terminates when a stopping criterion is met, such as a predefined number of episodes, a satisfactory policy, or a computational resource limit.

In essence, RL is a dynamic process in which an agent interacts with an environment,

learns from experiences, updates its policy or value estimates, and gradually hones its decision-making abilities. The specific algorithms and techniques may vary depending on the RL approach (e.g., Q-learning, policy gradient methods, actor-critic), but the fundamental flow remains consistent across RL paradigms.

Chapter 3

Literature Review and Contribution

This sections is designed for reviewing the developing concept of RL to get involved into QUIC environment.

3.1 Environment Of RL in Depth

Creating a reinforcement learning environment involves defining the state space, action space, reward system, and the overall interaction between the agent and the environment. Here's an outline of how such an environment can be designed:

State Space:

The state space represents the information available to the reinforcement learning agent. In the context of QUIC congestion control, the state space could include: Current network conditions: Bandwidth, latency, packet loss rate, and congestion indicators.

QUIC connection parameters:

Congestion window size, round-trip time, and number of unacknowledged packets.

History of past actions and states:

Agent's previous decisions and observations.

Action Space:

The action space defines the set of actions the agent can take. In the case of QUIC congestion control, possible actions could include: Adjusting the congestion window size: Increasing, decreasing, or maintaining the current window size. Adjusting the sending

rate: Increasing or decreasing the rate at which packets are sent.

Reward System:

The reward system provides feedback to the agent based on its actions and the resulting network performance. The reward system should incentivize the agent to maximize throughput, minimize packet loss, and maintain low latency. A possible reward system for evaluating QUIC congestion control could include: Positive reward for high throughput and low latency. Negative reward for high packet loss or excessive queuing delay. Penalty for aggressive behavior leading to unfair bandwidth utilization.

Interaction and Training Loop:

The agent interacts with the environment in an iterative training loop, where it observes the current state, takes actions, receives rewards, and updates its policy to improve its decision-making. The training **loop can be structured as follows:**
The agent observes the current state from the environment. Based on the observed state, the agent selects an action from the action space. The environment applies the chosen action to simulate the impact on the QUIC congestion control. The environment provides feedback in the form of rewards based on the resulting network performance. The agent updates its policy using the observed state, chosen action, and received reward to improve future decision-making. The process repeats for multiple iterations or episodes, gradually improving the agent's performance.

Training Data Collection:

To train the reinforcement learning agent effectively, a dataset needs to be collected. This can involve running simulations or conducting experiments in various network conditions, with different congestion scenarios and traffic patterns. The agent's actions, observed states, and the resulting rewards can be recorded during these training sessions. By defining the reinforcement learning environment with an appropriate state space, action space, reward system, and training loop, researchers can evaluate the QUIC protocol's congestion control using reinforcement learning algorithms. This environment enables the agent to learn and adapt its decision-making based on network feedback, ultimately improving the performance of the QUIC congestion control mechanism.

3.2 Developing Environment Of RL

Implementing a complete reinforcement learning system to evaluate the QUIC protocol's congestion control in Python requires integrating various components, including the reinforcement learning algorithm, the QUIC simulation environment, and the training loop. Given the complexity and size of the implementation, I'm developed Environment based on QUIC implementation based on python (aioquic). here is sample regarding different parts and its functionalities.

```
import gym  # For creating the reinforcement learning environment
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# Define the QUIC simulation environment as a Gym environment
class QUICEnvironment(gym.Env):
    def __init__(self):
        # Initialize the environment
        # Define state and action spaces, and reward system

    def reset(self):
        # Reset the environment to the initial state
        # Return the initial state

    def step(self, action):
        # Execute the action in the environment
        # Update the environment state based on the action
        # Calculate the reward based on the resulting #
        # network performance
        # Return the new state, reward, done flag, and
        # additional information

# Define the reinforcement learning agent
class QAgent:
    def __init__(self, state_size, action_size):
        # Initialize the agent
        # Define the neural network model for Q-learning
        # Define other parameters like epsilon-greedy
        # exploration, learning rate, etc.

    def act(self, state):
        # Choose an action based on the current state
        # Use the epsilon-greedy policy
        # for exploration and exploitation

    def train(self, state, action, reward, next_state, done):
        # Train the agent by updating the Q-values
        # based on the observed transition

# Initialize the QUIC environment and the agent
env = QUICEnvironment()
agent = QAgent(env.state_space, env.action_space)
```

```
# Training loop
for episode in range(num_episodes):
    state = env.reset()
    total_reward = 0

    for step in range(max_steps):
        # Choose an action based on the current state
        action = agent.act(state)

        # Execute the action in the environment
        next_state, reward, done, info = env.step(action)

        # Train the agent by updating the Q-values
        agent.train(state, action, reward, next_state, done)

        state = next_state
        total_reward += reward

        if done:
            break

    # Print episode statistics
    print(f"Episode: {episode+1}, Total Reward: {total_reward}")

# Evaluate the trained agent
total_rewards = []
num_eval_episodes = 10

for _ in range(num_eval_episodes):
    state = env.reset()
    total_reward = 0

    for _ in range(max_steps):
        action = agent.act(state)
        next_state, reward, done, _ = env.step(action)

        state = next_state
        total_reward += reward

        if done:
            break

    total_rewards.append(total_reward)
```

```
# Print evaluation results
average_reward = np.mean(total_rewards)
print(f"Average reward over {num_eval_episodes}")
```

3.2.1 gym library

The "gym" library mentioned is OpenAI Gym, a highly adopted Python library employed for the creation and assessment of reinforcement learning (RL) algorithms. This library furnishes a uniform and user-friendly interface for the formulation and interaction with RL environments, rendering it a favored selection within the RL community for researchers and developers alike. OpenAI Gym streamlines the development and evaluation of RL algorithms, establishing itself as an invaluable asset for individuals engaged in the discipline of reinforcement learning, whether they are researchers or practitioners.

During Development, I was using mainly Q-learning agent and the gym and stable-baselines3 libraries for defining the environment. Also I was leveraging socket programming in order to capture on-the-fly data from real network environment and QUIC implementation and train my model.

3.2.2 Stable-baselines3 library

Stable Baselines3 is a popular Python library used in reinforcement learning (RL) for developing, training, and evaluating RL algorithms. It's the successor to the earlier Stable Baselines library and is part of the OpenAI project. Stable Baselines3 offers a variety of RL algorithms, making it easier for researchers and practitioners to experiment with and apply RL in various domains.Stable Baselines3 simplifies RL experimentation by providing a high-quality, well-documented, and efficient framework. It's a valuable tool for researchers, engineers, and hobbyists interested in reinforcement learning, offering a wide range of algorithms and functionalities to support various RL tasks.

3.2.3 Developed RL's Environment

In this part, I would like to represent the implementation that I've developed in order to get connected to Quic implementation and capture the data and get them involved into RL process and train model based on data that are received and make action on real environment.Figure 3.1,3.1

This is Environment that has been developed:

```
import gymnasium as gym
import numpy as np
from gymnasium import spaces
import get_data
from aioquic.quic.recovery import QuicCongestionControl
```

```
quic_con = QuicCongestionControl()

class RL(gym.Env):
    """Custom Environment that follows gym interface."""

    metadata = {"render_modes": ["human"], "render_fps": 30}

    def __init__(self):
        super().__init__()
        # Define action and observation space
        # They must be gym.spaces objects
        # Example when using discrete actions:
        self.action_space = spaces.Discrete(3)
        self.observation_space = spaces.Box(low=0, high=255,
                                            shape=(5,2),dtype=np.float32)
        self.current_reward = 0.0
        self.prev_reward = 0.0
        self.reward = 0.0
        self.length = 60
        self.delta = 0.1
        self.reduction = QuicCongestionControl().reduction
        self.observation = None
        self.done = False

    def act_apply(self, action):
        self.reduction= self.reduction + (action * self.delta)

    def calculate_reward(self):
        average = np.mean(self.observation, axis=0)
        maximum = np.max(self.observation , axis = 0)
        reward = (average[0]/maximum[0]) - (average[1]/maximum[1])
        return reward

    def step(self, action):
        self.length -=1
        self.act_apply(action)
        self.current_reward = self.calculate_reward()
        if self.current_reward > self.prev_reward :
            self.reward += self.current_reward
            self.prev_reward = self.current_reward
        else:
            self.reward -= self.current_reward
            self.prev_reward = self.current_reward
```

```
        if self.length <=0:
            self.done = True
        else :
            self.done : False
        info = {}
        return self.observation, self.current_reward, self.done, info

    def reset(self, seed=None, options=None):
        self.done = False
        self.observation = np.array(get_data.recieve_data())
        return self.observation
```

Training model based on custom environment:

```
from env import RL
from stable_baselines3 import PPO,A2C
from stable_baselines3.common.vec_env import dummy_vec_env
from stable_baselines3.common.evaluation import evaluate_policy
import time
import os

models_dir = f"model-PPO-{int(time.time())}"
logdir = f"logs-PPO-{int(time.time())}"

if not os.path.exists(models_dir):
    os.mkdir(models_dir)
if not os.path.exists(logdir):
    os.mkdir(logdir)

env = RL()


episodes = 10
for episode in range(1, episodes+1 ):
    obs = env.reset()
    done = False
    score = 0

    while not done:
        action = env.action_space.sample()
        obs ,reward, done ,info  = env.step(action)
        score += reward
    print(f'Episode: {episode}, Score: {score}')
env.close()
```

```
def build_model(states,actions):
    model = Sequential()
    model.add(Dense(24,activation="relu",input_shape=states))
    # pass value to the deep learning model
    model.add(Dense(24,activation="relu"))
    model.add(Dense(actions,activation='linear'))
    return model


model = build_model(states,actions)
print(model.summary())
# del model

def build_agent(model, actions):
    policy = BoltzmannQPolicy()
    memory = SequentialMemory(limit=50000, window_length=1)
    dqn = DQNAgent(model=model, memory=memory, policy=policy,
                   nb_actions=actions, nb_steps_warmup=10
                   target_model_update=1e-2)
    return dqn

dqn = build_agent(model, actions)
dqn.compile(Adam(lr=1e-3), metrics=['mae'])
dqn.fit(env, nb_steps=50000, visualize=False, verbose=1)

scores = dqn.test(env, nb_episodes=100, visualize=False)
print(np.mean(scores.history['episode_reward']))
```

Chapter 4

Experimental/numerical evaluation

3

4.1 Methodology

This chapter outlines the methodology employed in conducting the research to compare the performance of HTTP and QUIC in diverse network conditions as well as optimize the performance of congestion control of the QUIC protocol in various network conditions using reinforcement learning techniques. The research methodology encompasses the simulation setup, lab infrastructure, and the procedures for evaluating the two protocols under various network conditions and optimization based on Reinforcement learning. The research approach consists of two primary phases: (1) Performance Evaluation and (2) Reinforcement Learning-based Optimization.

4.2 Simulation Environment

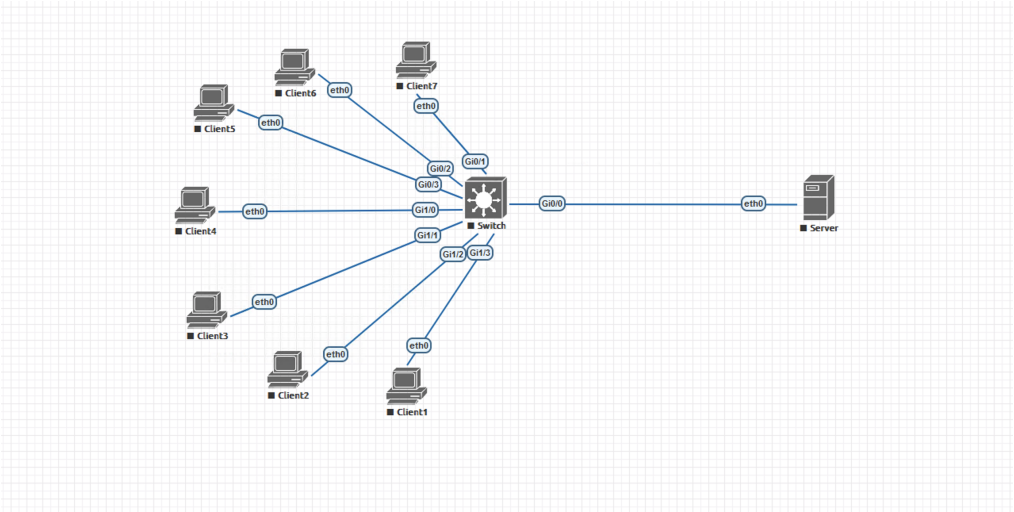
4.2.1 Selection of Network Simulation Software

To establish a controlled and replicable environment for network testing. In this study, i selected OMNET for its capability to accurately model various network conditions and its flexibility in customizing network parameters.

4.2.2 Simulation Configuration

i configured the network simulation software to replicate real-world network scenarios, including different network delays, latency levels, and packet loss rates. The chosen parameters were based on common network conditions encountered in today's internet, ensuring the relevance of our study.

Figure 4.1. Simulation Environment



4.2.3 Lab Setup and Hardware Infrastructure

In addition to simulation, i recognized the importance of conducting experiments in a physical lab environment. To facilitate this, i set up a dedicated lab with the following hardware components:

- HP DL860 prolight G4 server
- VMware ESXi virtualization
- linux OS
- Virtual Switch

4.2.4 Software Configuration

The lab's software infrastructure was designed to mirror the simulation environment as closely as possible. i installed and configured the necessary software components, including web servers, clients, and network monitoring tools. Special attention was given to ensuring that the lab setup was compatible with both HTTP and QUIC protocols.

4.3 Experimental Design

4.3.1 Selection of Test Cases.

i designed a series of test cases to evaluate the performance of HTTP and QUIC under different network conditions. These test cases include varying levels of:

- Network delay

- Latency
- Packet loss

4.3.2 Data Collection

For each test case, i collected data on several performance metrics, such as:

- Page load time
- Data transfer rate
- Reliability of data transmission

Data was gathered from both the simulation environment and the physical lab setup.

4.4 Performance Evaluation

4.4.1 Data Analysis

The collected data was analyzed using statistical techniques and performance evaluation metrics to compare the performance of HTTP and QUIC under different network conditions. i employed tools such as tcpdump,etc for this purpose.

4.4.2 Interpretation of Results

The results of the analysis were interpreted to draw meaningful conclusions regarding the performance of HTTP and QUIC in varying network conditions. i considered factors such as efficiency, reliability, and adaptability of the two protocols.

4.4.3 Comparative Metrics

To conduct a comprehensive comparison, i utilized a set of predefined metrics, including:

- Throughput
- Round-trip time (RTT)

4.4.4 Visualization

To facilitate a clearer understanding of the results, i employed visualization tools and techniques, such as graphs, charts, and diagrams, to represent the comparative performance of HTTP and QUIC across different network conditions.

4.5 Reinforcement Learning-based Optimization

4.5.1 Creation of Reinforcement Learning Environment

To optimize QUIC congestion control, i developed a specialized reinforcement learning environment. This environment was designed to closely resemble a real network environment and incorporated the actual QUIC protocol implementation. It allowed for the dynamic adjustment of QUIC's congestion control parameters in response to changing network conditions.

4.5.2 Data Collection

In this phase, i collected network data and statistics relevant to the QUIC protocol. Data sources included:

- Network topology and configuration information
- Real-time network performance metrics, such as RTT, throughput, and packet loss rates
- QUIC-specific metrics and protocol-level information

Network topology and configuration information Real-time network performance metrics, such as RTT, throughput, and packet loss rates QUIC-specific metrics and protocol-level information

4.5.3 Preprocessing

The collected data underwent preprocessing to make it suitable for use in the reinforcement learning framework. This involved data cleaning, transformation, and feature engineering to extract relevant information.

4.5.4 Environment Validation

To ensure that our reinforcement learning environment accurately represented real-world network conditions and QUIC behavior, i conducted validation experiments. These experiments included comparing the environment's performance against that of an actual QUIC implementation in a controlled lab environment.

4.6 Reinforcement Learning-based Optimization

4.6.1 Model Selection

I chose an appropriate reinforcement learning algorithm for the optimization task. Common algorithms used for congestion control optimization include Deep Q-Network (DQN), Proximal Policy Optimization (PPO), and Trust Region Policy Optimization (TRPO).

4.6.2 Model Architecture

I designed and implemented the reinforcement learning model architecture, including neural network structures and hyperparameters. The model was configured to take network conditions and QUIC parameters as input and provide optimal congestion control parameter adjustments as output.

4.6.3 Training

The model was trained using historical network data within the reinforcement learning environment. Training iterations were conducted, allowing the model to learn optimal parameter adjustments to improve network performance, especially in the presence of packet loss.Figure 4.2 , 4.3

4.6.4 Model Evaluation

The trained model's performance was rigorously evaluated through various experiments and scenarios, including different network conditions and packet loss scenarios. Performance metrics such as RTT, throughput, and stability were used to assess the effectiveness of the model in optimizing QUIC congestion control.

4.6.5 Conclusion

This chapter has outlined the methodology employed in the research to compare the performance of HTTP and QUIC in diverse network conditions. The simulation environment, lab setup, experimental design, data collection, and performance evaluation procedures were explained in detail. The next chapter will present the results of the study and discuss the findings in depth.

Figure 4.2. RL Environment

```python
import gymnasium as gym
import numpy as np
from gymnasium import spaces
import get_data
from aioquic.quic.recovery import QuicCongestionControl

quic_con = QuicCongestionControl()

class RL(gym.Env):
    """Custom Environment that follows gym interface."""

    metadata = {"render_modes": ["human"], "render_fps": 30}

    def __init__(self):
        super().__init__()
        # Define action and observation space
        # They must be gym.spaces objects
        # Example when using discrete actions:
        self.action_space = spaces.Discrete(3)
        # Example for using image as input (channel-first; channel-last also works):
        self.observation_space = spaces.Box(low=0, high=255,
                                            shape=(5,2), dtype=np.float32)
        self.current_reward = 0.0
        self.prev_reward = 0.0
        self.reward = 0.0
        self.length = 60
        self.delta = 0.1
        self.reduction = QuicCongestionControl().reduction
        self.observation = None
        self.done = False

    def act_apply(self, action):
        self.reduction= self.reduction + (action * self.delta)

    def calculate_reward(self):
        average = np.mean(self.observation, axis=0)
        maximum = np.max(self.observation , axis = 0)
        reward = (average[0]/maximum[0]) - (average[1]/maximum[1])
        # print(f"this is current observation {self.observation} , throughput avg us {average}, maximum is {maximum}")
        # print(f"this is reward {reward}")
        return reward

    def step(self, action):
        self.length -=1
        self.act_apply(action)
        self.current_reward = self.calculate_reward()
        if self.current_reward > self.prev_reward :
            self.reward += self.current_reward
            self.prev_reward = self.current_reward
        else:
            self.reward -= self.current_reward
            self.prev_reward = self.current_reward
        if self.length <=0:
            self.done = True
        else :
            self.done : False
        info = {}
        return self.observation, self.current_reward, self.done, info

    def reset(self, seed=None, options=None): #during initialization it may called
        self.done = False
        self.observation = np.array(get_data.recieve_data())
        return self.observation
```

Figure 4.3. Checking Environment and train model

```python
from env import RL
from stable_baselines3 import PPO,A2C
from stable_baselines3.common.vec_env import dummy_vec_env
from stable_baselines3.common.evaluation import evaluate_policy
import time
import os

models_dir = f"model-PPO-{int(time.time())}"
logdir = f"logs-PPO-{int(time.time())}"

if not os.path.exists(models_dir):
    os.mkdir(models_dir)
if not os.path.exists(logdir):
    os.mkdir(logdir)

env = RL()


episodes = 10
for episode in range(1, episodes+1 ):
    obs = env.reset()
    done = False
    score = 0

    while not done:
        action = env.action_space.sample()
        obs ,reward, done ,info  = env.step(action)
        score += reward
    print(f'Episode: {episode}, Score: {score}')
env.close()

```

```python
def build_model(states,actions):
    model = Sequential()
    model.add(Dense(24,activation="relu",input_shape=states)) # pass value to the deep learning model
    model.add(Dense(24,activation="relu"))
    model.add(Dense(actions,activation='linear'))
    return model


model = build_model(states,actions)
print(model.summary())
# del model

def build_agent(model, actions):
    policy = BoltzmannQPolicy()
    memory = SequentialMemory(limit=50000, window_length=1)
    dqn = DQNAgent(model=model, memory=memory, policy=policy,
                   nb_actions=actions, nb_steps_warmup=10, target_model_update=1e-2)
    return dqn

dqn = build_agent(model, actions)
dqn.compile(Adam(lr=1e-3), metrics=['mae'])
dqn.fit(env, nb_steps=50000, visualize=False, verbose=1)

scores = dqn.test(env, nb_episodes=100, visualize=False)
```

Chapter 5

Numerical results

This chapter encompasses a comprehensive presentation of the outcomes derived from our diverse set of tests. It commences with a comparison between two technologies, QUIC and HTTP, which offer varying levels of consistency, and scrutinizes their results across various metrics. Following this, i elucidate the consequences of employing Reinforcement Learning to fine-tune congestion control parameters and conduct an in-depth analysis of the merits and demerits of each approach, emphasizing the performance trade-offs involved. Additionally, within this section, you will find a series of experiments pertaining to migration. To provide a clear structure, our presentation of results comprises two distinct phases: the first phase entails the results of performance Comparison, while the second phase centers around the impact of Reinforcement Learning on enhancing QUIC.

5.1 Performance Comparison of QUIC and TCP

In this experiment, i conducted tests to assess how various network link parameters influence the throughput of both TCP and QUIC protocols. The primary objective of this experiment was to determine whether QUIC performs as well as or better than TCP by examining how these protocols respond to different network conditions.

To initiate the experiment, i established a controlled testing environment using Physical Environment. i then generated a 5M file for data transfer purposes. Subsequently, i employed a network configuration illustrated in Figure 4.1 as our reference setup. Within this environment, i executed the QUIC implementation and collected data on its performance.

An important procedural note is that i utilized iperf before initiating the client-side test. This step was necessary to capture network traffic data, allowing us to gather essential information for analysis. By conducting these tests and observations, i aimed to gain insights into how TCP and QUIC protocols operate and adapt in various network environments, ultimately determining if QUIC exhibits similar or superior performance compared to TCP.

In our experiment, each test case is characterized by a predefined set of network parameters. These parameters include the specific interval at which i modify one particular variable. Subsequently, i conduct a series of tests using both TCP and QUIC protocols within this controlled parameter environment. During these tests, i gather a variety of performance metrics, including measurements related to bandwidth and throughput. This comprehensive data collection allows us to assess how the two protocols perform under varying network conditions and analyze their respective capabilities in handling different parameter settings.

To carry out analogous tests with TCP, i took measures to ensure that the volume of data transmitted matched the file size utilized in the QUIC experiment. Furthermore, i maintained consistent usage of traffic shaping techniques in alignment with the methods previously employed during the QUIC testing phase. This systematic approach enabled us to directly juxtapose the performance of TCP and QUIC within a level playing field, facilitating a direct comparison of their behaviors and efficiencies across a spectrum of network parameter configurations.

5.1.1 Throughput comparison in different bandwidth

bandwidth test was performed by varying the bandwidth on all links between clients and server in my topology.Since there were multiple client hosts, I ran tests from each client and averaged the throughput results across all clients to plot the final bandwidth vs throughput graph.

In the bandwidth test (Figure 5.1), TCP throughput increased linearly as available bandwidth increased from 2Mbps to 100Mbps. QUIC was able to match the throughput of TCP up to an available bandwidth of 20Mbps. At higher bandwidths above 20Mbps, QUIC throughput plateaued and no longer scaled linearly.

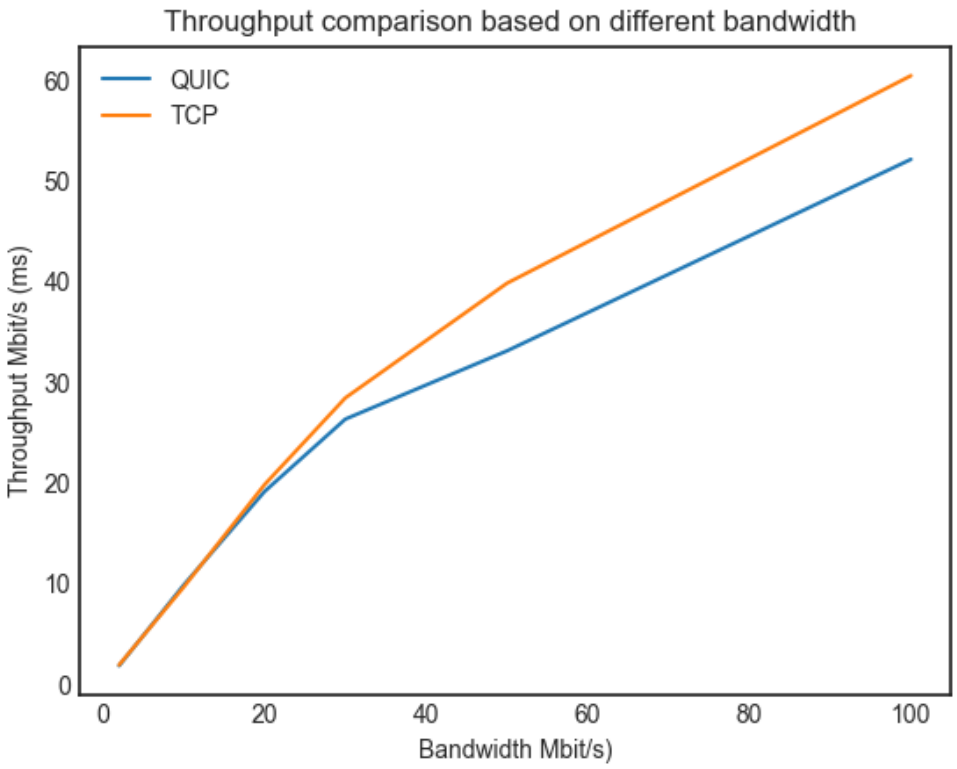
The maximum throughput achieved for QUIC was 52.16 Mbit/s, compared to 60.47 Mbit/s for TCP. The minimum throughput at 2Mbps bandwidth was 1.85 Mbit/s and 1.93 Mbit/s for QUIC and TCP respectively.

This indicates that QUIC is able to fully utilize the available bandwidth up to around 20Mbps, but does not increase its throughput as aggressively as TCP at higher bandwidths.

5.1.2 Throughput comparison in different Loss

The loss test revealed that QUIC is more resilient than TCP when it comes to packet loss. As the packet loss rate increased from 0% to 20%, the throughput for both QUIC and TCP dropped sharply. However, QUIC was able to achieve significantly higher throughput than TCP at all loss rates above 5%.

Figure 5.1. QUIC vs TCP - Bandwidth Comparison

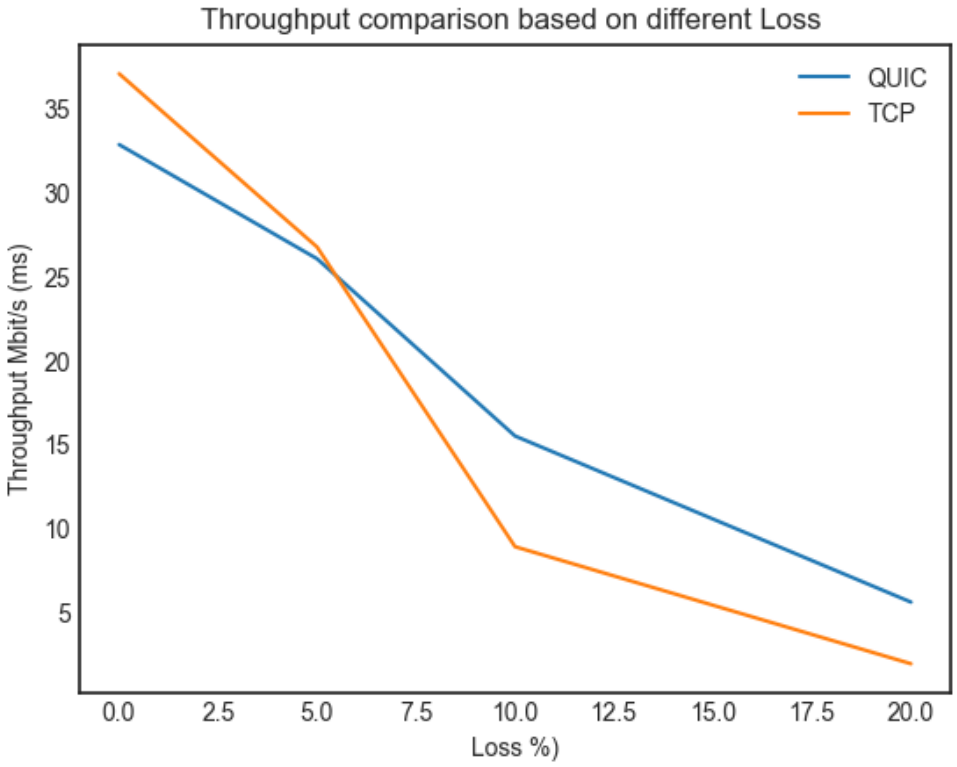


For example (Figure 5.2), at 20% packet loss, QUIC sustained a throughput of 7 Mbps while TCP throughput dropped to just 2 Mbps. This indicates that QUIC is better able to handle lossy networks and maintain usable performance despite heavy packet loss. The reasons could include QUIC's ability to recover packets purely at the application layer or differences in how the congestion control algorithms react to loss events.

In general, I would highlight:

- QUIC consistently outperformed TCP at high loss
- Quantify the throughput difference at a very lossy point (e.g. 7 Mbps vs 2 Mbps at 20% loss)
- Discuss possible reasons for QUIC's better loss resilience
- Note the sharp decline in throughput for both as loss passes 5%

Figure 5.2. QUIC vs TCP - Different Loss



5.1.3 Throughput comparison in different Delay

The latency test revealed that both QUIC and TCP throughput degraded gradually as the round-trip time (RTT) increased from 80ms to 800ms. However, QUIC was able to achieve marginally higher throughput than TCP at some tested delay values.

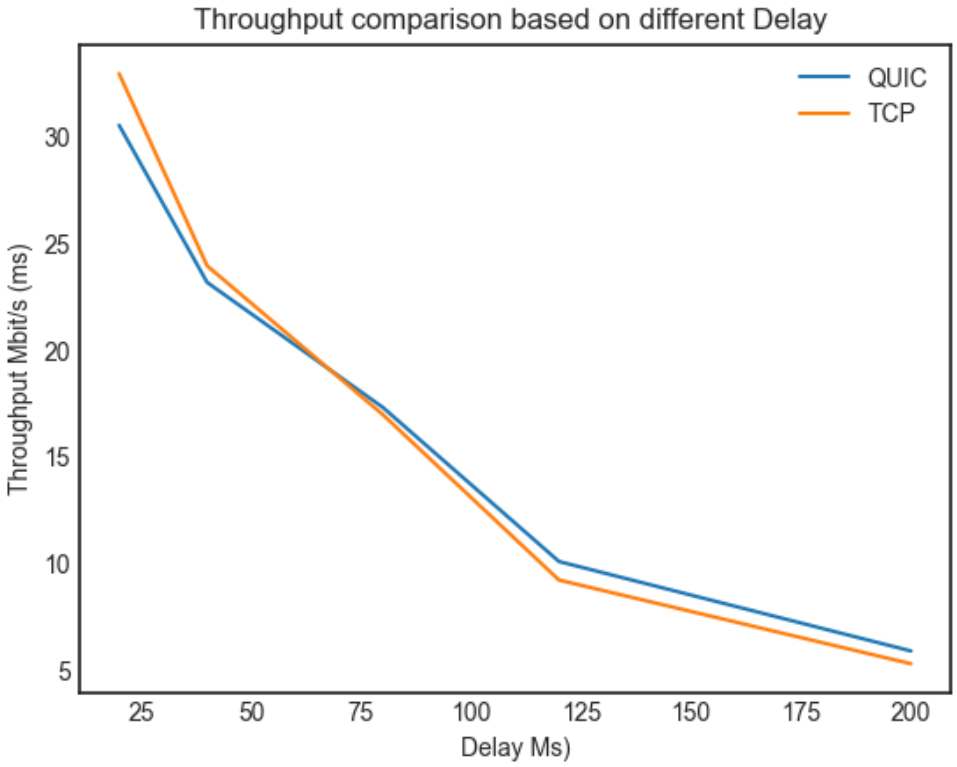
For example based on figure 5.3, at 200ms Delay, QUIC sustained a throughput of 5.9 Mbps while TCP throughput was 5.3 Mbps. The throughput decline was less severe compared to the packet loss tests, indicating that both protocols are more resilient to delay rather than loss.

The reasons for QUIC's slightly better delay performance could include its ability to better utilize available bandwidth during the handshake phase due to 0-RTT connection resumption. However, the throughput difference was minor, suggesting both protocols are capable of handling increased network latency.

In summary:

- Emphasize the gradual decline in throughput vs sharp drop for loss tests

Figure 5.3. QUIC vs TCP - Different Delay



- QUIC had a slight edge over TCP at all delays
- Quantify small throughput difference at high delay (e.g. 5.9 Mbps vs 5.3 Mbps)
- Note that both protocols are fairly delay resistant

5.2 RL optimization on QUIC Congestion Control

This phase of the experiment aims to investigate how the performance of the protocol is impacted by varying parameters within the congestion control algorithm. Since congestion control predominantly influences congestion window behavior.Building upon our prior exploration of the Cubic protocol in previous Chapter, it is important to focus on the two key parameters, C and β, that govern the algorithm's behavior. Let's delve into the influence of β and C on the curve, as they play significant roles in shaping the characteristics of QUIC.

I will demonstrate the impact of QUIC's reduction factor on congestion control performance after losses occur. The goal is to analyze how the different beta values affect

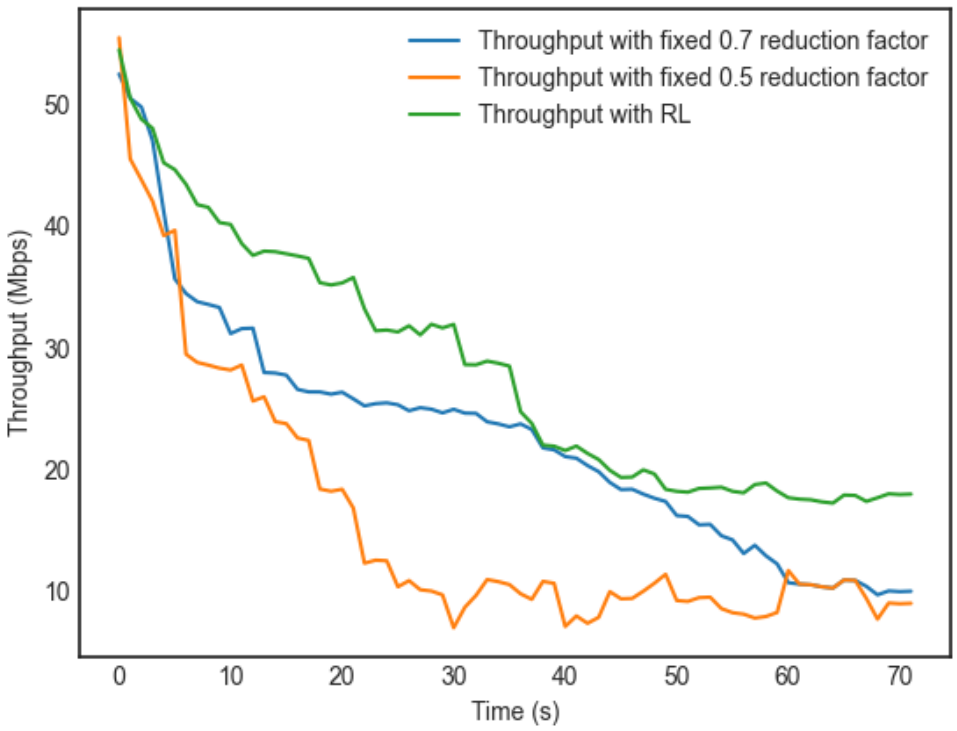
overall behavior in response to congestion signals.

5.2.1 Tuning performance based on Reduction Factor

I conducted a performance test for QUIC under specific network conditions, where I utilized a 100 Mbit/s bandwidth with a 5% packet loss rate. The evaluation primarily focused on throughput, and I experimented with two static reduction factor (beta) values: 0.5 and 0.7. Additionally, I employed a Reinforcement Learning (RL) model that dynamically adjusted the reduction factor to enhance performance.

Regarding the static values (Figure 5.4), I observed distinct behaviors for reduction factors 0.5 and 0.7. Reduction factor 0.7 exhibited a gradual and smooth decrease in throughput over time until it reached a stable condition. In contrast, reduction factor 0.5 displayed a more aggressive approach, with a sharp initial drop in throughput to achieve stability in a shorter timeframe. However, it exhibited greater fluctuations over time compared to 0.7. Both static values maintained their respective behaviors consistently throughout the testing period..

Figure 5.4. Comparison Throughput based on static Reduction factors and RL tuning

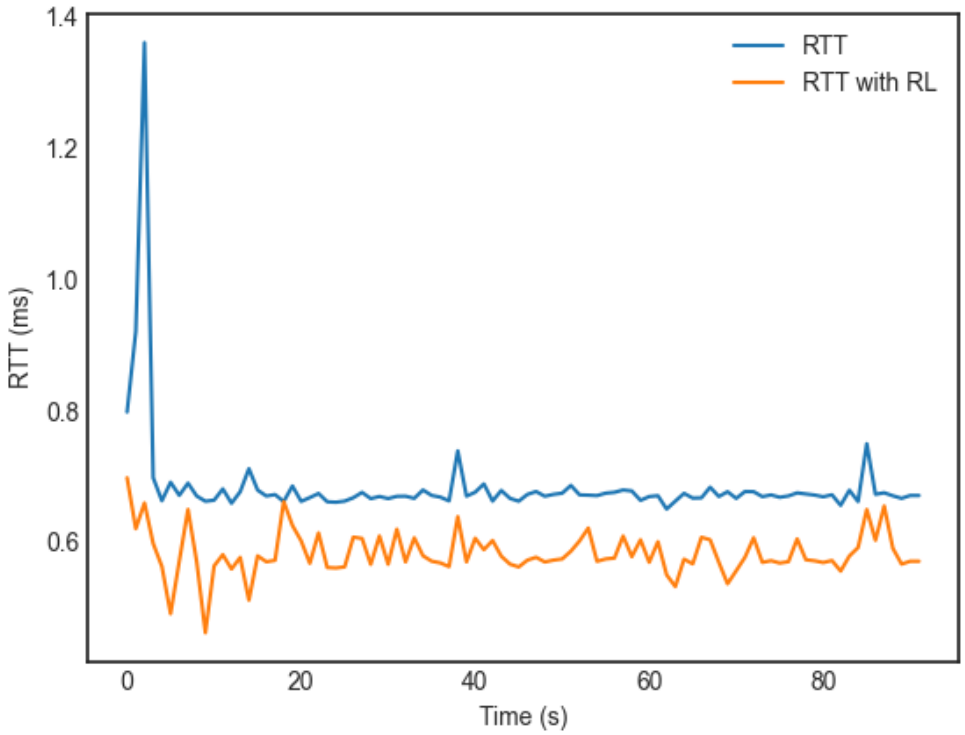


On the RL side (Figure 5.4), the dynamic adjustment of the reduction factor based on real-time network conditions led to gradual or significant throughput decreases, depending on the conditions at each moment. Although it took a longer time for the RL model to reach a stable state, it consistently outperformed the static reduction factor values at any given moment, delivering better overall performance throughout the test.

In addition to throughput, I also monitored Round-Trip Time (RTT) (Figure 5.5). I observed that the RL model was effective at keeping RTT as low as possible compared to the static reduction factor values, indicating that the RL approach had a positive impact on reducing network latency.

In summary, the dynamic nature of the RL model allowed it to adapt to changing network conditions, resulting in improved performance, albeit with a longer convergence time compared to the static reduction factor values.

Figure 5.5. Comparison RTT based on Reduction factor and RL Tuning



Chapter 6

Conclusion

In summary, this thesis has made substantial contributions to the field of network protocol evaluation, particularly in the context of QUIC and TCP implementations. The primary focus of this research has been on network performance assessment and congestion handling capabilities. Additionally, a novel approach involving the use of Reinforcement Learning (RL) to improve QUIC congestion control was explored.

One of the key highlights of this work has been the comprehensive testing of QUIC and TCP implementations using a diverse set of metrics, with a particular emphasis on the evaluation of throughput stability. These evaluations have shed light on the strengths and limitations of both protocols. QUIC has demonstrated its ability to overcome certain limitations associated with TCP, such as head-of-line blocking and reliance on the operating system's TCP version. While various new features of QUIC have been briefly mentioned in this thesis, they offer promising avenues for further exploration.

The performance evaluations have revealed that QUIC behaves similarly to TCP in low-delay, packet-loss-free environments. Differences primarily stem from the slower Congestion Window (CWND) growth rate in QUIC. These tests have also confirmed that QUIC excels in scenarios with a low probability of packet loss, aligning with its design objectives.

The thesis further delves into the analysis of changing congestion control parameters, both statically and dynamically, through RL. The development of an RL environment closely aligned with the real-world lab environment allowed for training models that could dynamically adjust metrics, evaluate Throughput, and monitor Round-Trip Time (RTT). The results demonstrated that dynamic metric adjustments led to improved performance, particularly in the presence of unstable network conditions.

It is important to acknowledge that results obtained in isolated and emulated environments should be interpreted with caution, as real-world networks are influenced by various external factors and concurrent traffic. Validation through comparisons with real

network tests is a crucial next step. Furthermore, a deeper analysis is required to understand the specific circumstances under which protocol behavior and parameters should be adjusted to align more closely with TCP. This research serves as a valuable step towards refining the evaluation methodologies for congestion control protocols, with the hope of enhancing our understanding of these critical networking components.

Bibliography

[1] https://jacobianengineering.com/blog/2016/11/1543/.

[2] https://vasexperts.com/blog/functionality/from-tcp-to-quic/.

[3] https://www.v7labs.com/blog/deep-reinforcement-learning-guide.

[4] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[5] Mike Bishop et al. Hypertext transfer protocol version 3 (http/3). *Internet Engineering Task Force, Internet-Draft draft-ietf-quic-http-34*, 2021.

[6] C Cimpanu. Google creates new algorithm for handling tcp traffic congestion control, 2016.

[7] Monia Ghobadi, Yuchung Cheng, Ankur Jain, and Matt Mathis. Trickle: Rate limiting {YouTube} video streaming. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 191–196, 2012.

[8] Jana Iyengar and Martin Thomson. Rfc 9000 quic: A udp-based multiplexed and secure transport. *Omtermet Emgomeeromg Task Force*, 2021.

[9] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[10] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. Taking a long look at quic: an approach for rigorous evaluation of rapidly evolving transport protocols. In *Proceedings of the 2017 Internet Measurement Conference*, pages 290–303, 2017.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[12] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[13] Mohammad Shehab, Ahamad Tajudin Khader, and Mohammad Alia. Enhancing cuckoo search algorithm by using reinforcement learning for constrained engineering optimization problems. pages 812–816, 04 2019.

[14] Tanya Shreedhar, Rohit Panda, Sergey Podanev, and Vaibhav Bajpai. Evaluating quic performance over web, cloud storage, and video workloads. *IEEE Transactions on Network and Service Management*, 19(2):1366–1381, 2021.

[15] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[16] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[17] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

[18] Vibhore Tyagi, Sachi Pandey, and Tarun Kumar. A survey of tcp congestion control algorithm in wireless network: Bic and cubic. 01 2014.

[19] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[20] Alexander Yu and Theophilus A Benson. Dissecting performance of production quic. In *Proceedings of the Web Conference 2021*, pages 1157–1168, 2021.