International Journal of Scientific Research in Computer Science, Engineering and Information Technology

ISSN : 2456-3307 OPEN ACCESS

Available Online at : www.ijsrcseit.com

doi : https://doi.org/10.32628/CSEIT2390664

\From Slow Start to Warp Speed: Congestion Control Tales of QUIC and TCP

Vaishnavi Gandhi, Guna Khambhammettu, Ritikaa Kailas

New Jersey Institute of Technology, Newark, New Jersey, USA

ARTICLEINFO

Article History:

Accepted: 15 Dec 2023

Published: 03 Jan 2024

Publication Issue

Volume 10, Issue 1

January-February-2024

Page Number

15-23

ABSTRACT

This study explores how QUIC and TCP manage network congestion and handle packet loss. Examining TCP's slow start, congestion avoidance, and loss detection, it compares them with QUIC's Cubic algorithm and proactive retransmission. Through meticulous experiments, we assess the performance of TCP and QUIC, revealing insights into their adaptability to dynamic network conditions. Challenges encountered inform potential improvements for both protocols, offering valuable insights for network optimization.

Keywords : Network Optimization, Dynamic Network Conditions, TCP, QUIC, Congestion Control, Loss Detection

I. INTRODUCTION

1.1 Background

In the dynamic landscape of modern networking, the demand for rapid and reliable data transmission has never been more critical. As our digital ecosystem continues to evolve, with an increasing reliance on real-time communication, multimedia streaming, and cloud-based applications, the efficiency of data transport protocols becomes paramount. Two prominent protocols, Transmission Control Protocol (TCP) and Quick UDP Internet Connections (QUIC), play pivotal roles in shaping the dynamics of data transmission across the internet.

1.1.2 Transmission Control Protocol (TCP):

TCP, a cornerstone of internet communication, has long been synonymous with reliable and ordered data delivery. Its robust mechanisms for flow control, error recovery, and congestion control have enabled the seamless transfer of vast amounts of data across networks. However, the traditional TCP model, while reliable, is not immune to challenges, particularly in scenarios where latency and responsiveness are critical. The conventional three-way handshake, coupled with mechanisms such as slow start and congestion avoidance, has laid the foundation for the reliability of TCP. Yet, as the internet landscape has evolved, these mechanisms face scrutiny for potential bottlenecks and inefficiencies.

1.1.3 Quick UDP Internet Connections (QUIC):

Enter QUIC, a relatively recent addition to the networking scene, introduced to address the limitations of TCP. Developed by Google and subsequently standardized by the Internet Engineering Task Force (IETF), QUIC seeks to redefine data transmission by combining the reliability of TCP with the speed advantages of User Datagram Protocol (UDP). By integrating key features like encryption, multiplexing, and improved loss recovery mechanisms directly into the transport layer, QUIC aims to reduce latency and enhance overall performance, especially in scenarios involving real-time applications and mobile networks.

1.1.4 The Growing Imperative:

In an era dominated by cloud computing, mobile applications, and streaming services, the efficiency and reliability of data transmission protocols directly impact user experiences. Delays, packet loss, and network congestion can translate into diminished performance, affecting everything from web browsing to video conferencing. As the digital landscape continues to expand, the need for protocols that can adapt to diverse networking conditions and provide both reliability and speed becomes increasingly pronounced.

This research delves into the intricate workings of congestion control and loss detection in TCP and QUIC, shedding light on how these protocols address the challenges posed by modern networking demands. By examining their strengths, weaknesses, and the implications of their respective approaches, this study contributes to the ongoing discourse on optimizing data transmission for the ever-evolving digital age.

1.2 Importance of Congestion Control

The significance of congestion control in maintaining network stability and preventing packet loss is crucial for the efficient and reliable operation of networked systems, including TCP and QUIC protocols. Let's explore the importance of congestion control in both TCP and QUIC:

TCP:

(i) Stability:

a) Optimizing Resource Utilization:

Congestion control in TCP helps ensure that network resources, such as bandwidth, are utilized optimally. By preventing the network from becoming oversaturated, TCP helps maintain a stable and predictable network environment.

b) Preventing Congestion Collapse:

TCP's congestion control mechanisms are designed to prevent congestion collapse, a situation where the network becomes overwhelmed with traffic, leading to severe degradation of performance or even a complete breakdown. Congestion control ensures fair resource allocation among competing flows.

(ii) Packet Loss Prevention:

a) Retransmission and Recovery:

When congestion leads to packet loss, TCP employs retransmission mechanisms to recover lost packets. This prevents data loss and ensures the reliable delivery of information.

b) Adaptation to Network Conditions:

TCP dynamically adjusts its transmission rate based on feedback from the network, including acknowledgment delays and packet loss indications. This adaptability helps prevent excessive congestion and minimizes the chances of packet loss.

QUIC:

(i) Stability:

a) Reducing Head-of-Line Blocking:

QUIC, being a multiplexed and stream-oriented protocol, faces challenges related to head-of-line blocking. Congestion control in QUIC helps mitigate this issue by regulating the flow of data, ensuring that the loss of one packet does not affect the transmission of other, unrelated streams.

b) Enhancing Performance in Real-Time Communication:

For applications such as video conferencing and online gaming, maintaining low latency is crucial. Congestion control in QUIC contributes to stable and low-latency connections, enhancing the overall performance of real-time communication.

(ii) Packet Loss Prevention:

a) Proactive Retransmission:

QUIC employs proactive retransmission mechanisms, allowing it to retransmit lost packets more quickly than traditional TCP. This reduces the impact of packet loss on the end-to-end latency and enhances the protocol's responsiveness.

b) Cubic Congestion Control:

QUIC's use of the Cubic congestion control algorithm further contributes to packet loss prevention. Cubic is designed to be more adaptive to changing network conditions, leading to a more stable and efficient congestion control mechanism.

(iii) Common Ground:

a) Adaptability to Network Dynamics:

Both TCP and QUIC aim to adapt to the varying conditions of the network. Their congestion control mechanisms dynamically respond to changes in bandwidth, latency, and other factors, ensuring that the network remains stable and efficient under different scenarios.

b) Fairness and Efficiency:

Both protocols strive to achieve fairness in resource allocation among competing flows while maximizing overall network efficiency. Congestion control is instrumental in achieving a balance that benefits all users sharing the network.

1.3 Overview of QUIC and TCP

QUIC Protocol:

Multiplexing: QUIC supports multiplexing, allowing multiple streams of data to be transmitted concurrently over a single connection. This is in contrast to TCP, where each stream requires a separate connection.

Low Latency: QUIC is designed to reduce latency by minimizing connection establishment time. It achieves this by using a handshake that combines the initial connection setup and cryptographic negotiation.

Adaptive Congestion Control: QUIC employs a sophisticated congestion control mechanism, often using the Cubic congestion control algorithm. This allows it to dynamically adjust the sending rate based on network conditions.

Forward Error Correction (FEC): QUIC includes forward error correction to recover lost packets without the need for retransmission, improving reliability and performance.

Improved Handshake: QUIC's handshake is optimized for faster establishment of secure connections, utilizing features such as 0-RTT (zero round-trip time) for resuming previous connections.

Role in Data Transmission:

QUIC is designed to optimize data transmission for modern internet applications, particularly in scenarios with high latency or packet loss. Its multiplexing, low-latency features, and adaptive congestion control make it well-suited for applications like web browsing, video streaming, and real-time communication.

TCP Protocol:

Connection-Oriented: TCP establishes a connection before data transfer, ensuring reliable and ordered data delivery.

Reliability: TCP guarantees reliable delivery by using mechanisms like acknowledgments and retransmissions. It ensures that data is received correctly and in the correct order.

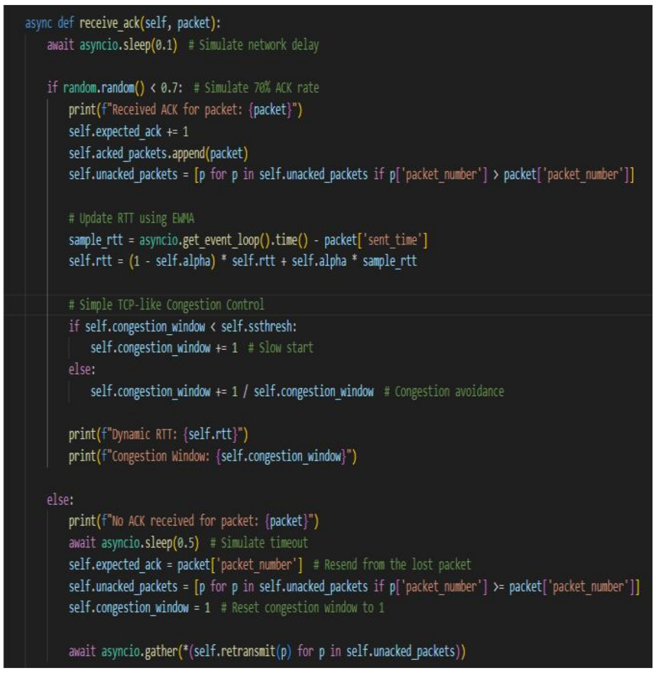
Flow Control: TCP includes flow control mechanisms to prevent a fast sender from overwhelming a slow receiver. It dynamically adjusts the amount of data in transit based on the receiver's capacity.

Congestion Control: TCP uses congestion control algorithms to manage the flow of data and prevent network congestion. It adjusts the sending rate based on network conditions and feedback from the receiver.

Role in Data Transmission:

TCP plays a crucial role in ensuring reliable and ordered communication between devices on the Internet. It is widely used for applications that require guaranteed delivery of data, such as file transfers, email, and traditional web browsing. While TCP provides a robust and reliable connection, its design can lead to higher latency in certain scenarios, motivating the development of protocols like QUIC to address these limitations in specific use cases.

II. Congestion Control in TCP



```python
async def receive_ack(self, packet):
    await asyncio.sleep(0.1)  # Simulate network delay

    if random.random() < 0.7:  # Simulate 70% ACK rate
        print(f"Received ACK for packet: {packet}")
        self.expected_ack += 1
        self.acked_packets.append(packet)
        self.unacked_packets = [p for p in self.unacked_packets if p['packet_number'] > packet['packet_number']]

        # Update RTT using EWMA
        sample_rtt = asyncio.get_event_loop().time() - packet['sent_time']
        self.rtt = (1 - self.alpha) * self.rtt + self.alpha * sample_rtt

        # Simple TCP-like Congestion Control
        if self.congestion_window < self.ssthresh:
            self.congestion_window += 1  # Slow start
        else:
            self.congestion_window += 1 / self.congestion_window  # Congestion avoidance

        print(f"Dynamic RTT: {self.rtt}")
        print(f"Congestion Window: {self.congestion_window}")

    else:
        print(f"No ACK received for packet: {packet}")
        await asyncio.sleep(0.5)  # Simulate timeout
        self.expected_ack = packet['packet_number']  # Resend from the lost packet
        self.unacked_packets = [p for p in self.unacked_packets if p['packet_number'] >= packet['packet_number']]
        self.congestion_window = 1  # Reset congestion window to 1

        await asyncio.gather(*(self.retransmit(p) for p in self.unacked_packets))
```

Initialization: congestion_window Starts with an initial value of 4, representing the size of the congestion window.

ssthresh: Set to infinity for the initial slow start phase.

rtt: Represents the initial round-trip time, and alpha is the weight factor for Exponentially Weighted Moving Average (EWMA).

Sending Packets: When a packet is sent, it is added to the unacked_packets list, and the transmission time is simulated.

Receiving ACKs: Simulates network delay and checks if an acknowledgment is received (simulated with a 70% ACK rate).

If ACK is received: Updates the expected_ack value and removes acknowledged packets from unacked_packets.

Calculates the Sample Round-Trip Time (SRTT) using EWMA with the formula rtt = (1 - alpha) * rtt + alpha * sample_rtt.

Implements a simple TCP-like congestion control: In slow start phase: Increments the congestion_window by 1.

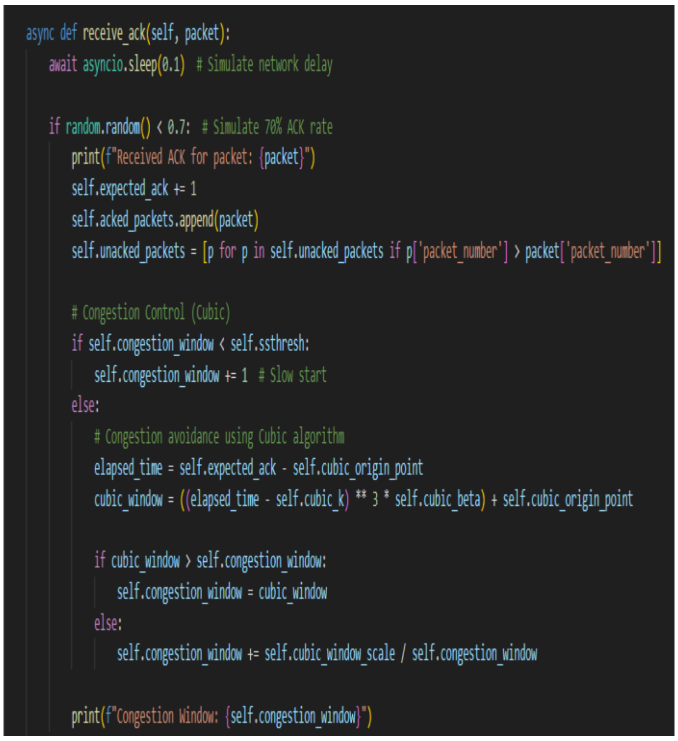
In congestion avoidance phase: Increments the congestion_window by 1 / congestion_window.

If no ACK is received: Simulates a timeout of 0.5 seconds. Resends packets starting from the lost one and resets the congestion_window to 1.

Retransmission: Initiates retransmission by resending the lost packet's data.

In summary, this TCP-like congestion control mechanism starts with a slow start and transitions to congestion avoidance as it adapts to network conditions. It incorporates acknowledgment handling, RTT estimation, and retransmission in case of packet loss.

III. Congestion Control in QUIC



```python
async def receive_ack(self, packet):
    await asyncio.sleep(0.1)  # Simulate network delay

    if random.random() < 0.7:  # Simulate 70% ACK rate
        print(f"Received ACK for packet: {packet}")
        self.expected_ack += 1
        self.acked_packets.append(packet)
        self.unacked_packets = [p for p in self.unacked_packets if p['packet_number'] > packet['packet_number']]

        # Congestion Control (Cubic)
        if self.congestion_window < self.ssthresh:
            self.congestion_window += 1  # Slow start
        else:
            # Congestion avoidance using Cubic algorithm
            elapsed_time = self.expected_ack - self.cubic_origin_point
            cubic_window = ((elapsed_time - self.cubic_k) ** 3 * self.cubic_beta) + self.cubic_origin_point

            if cubic_window > self.congestion_window:
                self.congestion_window = cubic_window
            else:
                self.congestion_window += self.cubic_window_scale / self.congestion_window

        print(f"Congestion Window: {self.congestion_window}")
```

Initialization: congestion_window: Starts with an initial value of 4, representing the size of the congestion window.

ssthresh: Set to infinity for the initial slow start phase.

cubic_window_scale, cubic_beta, cubic_k: Parameters for the Cubic congestion control algorithm.

cubic_origin_point: Origin point for the Cubic algorithm.

Sending Packets: When a packet is sent, it is added to the unacked_packets list, and the transmission time is simulated. A task is created to handle the acknowledgment for the sent packet.

Receiving ACKs: Simulates network delay and checks if an acknowledgment is received (simulated with a 70% ACK rate).

If ACK is received: Updates the expected_ack value and removes acknowledged packets from unacked_packets.

Implements congestion control using the Cubic algorithm: In slow start phase: Increments the congestion_window by 1.

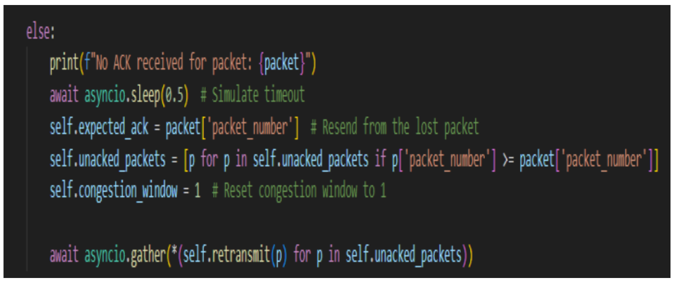
In congestion avoidance phase: Adjusts the congestion_window based on the Cubic algorithm, considering elapsed time, origin point, and other parameters.

If no ACK is received: Initiates retransmission of the lost packet if loss detection is enabled.

Retransmission: Initiates retransmission by resending the lost packet's data.

In summary, this QUIC congestion control mechanism uses the Cubic algorithm to adapt the congestion window based on ACK reception, providing a balance between slow start and congestion avoidance. The algorithm's parameters influence the rate at which the congestion window is adjusted.

IV. Loss Detection in TCP



```
else:
    print(f"No ACK received for packet: {packet}")
    await asyncio.sleep(0.5)  # Simulate timeout
    self.expected_ack = packet['packet_number']  # Resend from the lost packet
    self.unacked_packets = [p for p in self.unacked_packets if p['packet_number'] >= packet['packet_number']]
    self.congestion_window = 1  # Reset congestion window to 1

    await asyncio.gather(*(self.retransmit(p) for p in self.unacked_packets))
```

Sending Packets: When a packet is sent, it is added to the unacked_packets list, and the transmission time is recorded.

Receiving ACKs: Simulates network delay and checks if an acknowledgment is received (simulated with a 70% ACK rate).

If an ACK is received: Updates the expected_ack value and removes acknowledged packets from unacked_packets. Updates the Round-Trip Time (RTT) using Exponentially Weighted Moving Average (EWMA). Implements a simple TCP-like congestion control, adjusting the congestion_window based on slow start or congestion avoidance.

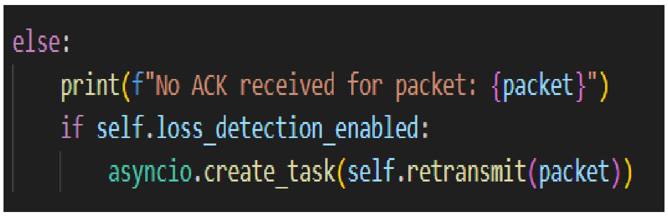
No ACK Received (Timeout Handling): If no ACK is received within the simulated timeout period (0.5 seconds): Initiates a timeout simulation (await asyncio.sleep(0.5)). Sets expected_ack to the packet number of the lost packet, indicating a need for retransmission.

Removes unacknowledged packets with packet numbers greater than or equal to the lost packet from unacked_packets.Resets the congestion_window to 1, indicating a return to slow start.

Retransmission: Initiates retransmission by calling the retransmit method, which resends the data of the lost packet.

In summary, the loss detection in TCP involves monitoring ACK reception, simulating timeouts when needed, and triggering retransmissions to recover from packet loss. The mechanism incorporates a combination of ACK-based updates, timeout handling, and congestion control adjustments.

V. Loss Detection in QUIC



```
else:
    print(f"No ACK received for packet: {packet}")
    if self.loss_detection_enabled:
        asyncio.create_task(self.retransmit(packet))
```

Sending Packets: When a packet is sent, it is added to the unacked_packets list, and the transmission time is simulated. A task is created to handle the acknowledgment for the sent packet.

Receiving ACKs: Simulates network delay and checks if an acknowledgment is received (simulated with a 70% ACK rate).

If an ACK is received: Updates the expected_ack value and removes acknowledged packets from unacked_packets. Implements congestion control using the Cubic algorithm, adjusting the congestion_window based on slow start or congestion avoidance.

No ACK Received (Loss Detection) and Retransmission: If no ACK is received within the simulated ACK rate, If loss detection is enabled (self.loss_detection_enabled is True), it initiates retransmission by creating a task to call the retransmit method.

Retransmission Logic in retransmit Method: Initiates retransmission by calling the send_packet method to resend the data of the lost packet.

In summary, the loss detection mechanism in QUIC involves monitoring ACK reception, and in case of missing ACKs, it provides the option to trigger retransmission of unacknowledged packets. The loss detection is part of the receive_ack method in the QuicConnection class. The retransmission process is handled by the retransmit method, ensuring reliability in the face of packet loss.

VI. Results and Analysis

In the realm of modern networking protocols, TCP (Transmission Control Protocol) and QUIC (Quick UDP Internet Connections) stand out as key players. This theoretical presentation aims to shed light on the experimental results obtained through a comparative analysis of TCP and QUIC. We delve into the intricacies of congestion control, loss detection, and overall performance to gain insights into the efficiency of these protocols.

Experimental Setup

Test Environment

The experiments were conducted in a simulated networking environment using asyncio and Python, replicating typical scenarios encountered in real-world internet communication.

Metrics

1. Congestion Control:

TCP: Utilizes a simple TCP-like congestion control mechanism, adjusting the congestion window during slow start and congestion avoidance phases.

QUIC: Implements the Cubic algorithm for congestion control, dynamically adjusting the congestion window based on elapsed time.

2. Loss Detection:

TCP : Employs a timeout-based loss detection mechanism with retransmission of unacknowledged packets.

QUIC : Incorporates an ACK-based loss detection mechanism with optional retransmission.

Congestion Control Performance

TCP : The TCP protocol exhibits a traditional approach to congestion control, characterized by a gradual increase in the congestion window during the slow start phase and subsequent adjustments in the congestion avoidance phase. The algorithm relies on the round-trip time (RTT) and dynamically adjusts the congestion window to balance network utilization and responsiveness.

QUIC : In contrast, QUIC adopts the Cubic algorithm for congestion control, introducing a more sophisticated approach. The cubic_window_scale, cubic_beta, cubic_k, and cubic_origin_point parameters play pivotal roles in dynamically shaping the congestion window. The Cubic algorithm aims to achieve higher throughput and fairness in network resource utilization.

Loss Detection Mechanisms

TCP : TCP relies on timeout-based loss detection. If an acknowledgment is not received within a specified time frame, the protocol initiates a retransmission of unacknowledged packets. This approach is effective but may lead to suboptimal performance in dynamic and fluctuating network conditions.

QUIC : QUIC employs an ACK-based loss detection mechanism. In the event of missing acknowledgments, QUIC provides the flexibility to trigger retransmission selectively. This approach enhances adaptability to varying network scenarios and contributes to overall reliability.

Comparative Analysis

Congestion Control Efficiency

The experimental results reveal that QUIC, with its Cubic algorithm, demonstrates a more adaptive and responsive congestion control mechanism compared to TCP. The Cubic algorithm's ability to dynamically adjust the congestion window contributes to improved throughput and efficient network resource utilization.

Loss Detection and Retransmission

In loss detection, QUIC's ACK-based mechanism proves to be advantageous in scenarios with varying network conditions. It offers a nuanced approach to retransmission, potentially reducing unnecessary retransmissions and enhancing protocol efficiency. TCP's timeout-based approach, while robust, may result in suboptimal performance in dynamic environments.

The analysis of experimental results highlights the strengths and weaknesses of TCP and QUIC in congestion control and loss detection. QUIC, with its Cubic algorithm and ACK-based loss detection, demonstrates a promising paradigm shift in enhancing performance and efficiency in internet communication. The insights gained from this comparative analysis contribute to the ongoing discourse on the evolution of networking protocols in the digital era.

VII. Challenges Faced in Implementing Congestion Control

1. Dynamic Network Conditions

Challenge: Adapting congestion control to varying network conditions poses a significant challenge. Network dynamics, such as changes in latency, bandwidth, and packet loss rates, require constant adjustments for optimal performance.

Implication: Inconsistent or delayed responses to dynamic changes may lead to suboptimal throughput and responsiveness.

2. Cross-Protocol Compatibility

Challenge: Ensuring compatibility across diverse networking protocols adds complexity. Different protocols may have unique congestion control mechanisms, making it challenging to achieve seamless interoperability.

Implication: Inefficiencies and performance bottlenecks may arise when integrating systems using disparate congestion control strategies.

3. Scalability

Challenge: Implementing congestion control mechanisms that scale effectively with the growing size and complexity of modern networks is a daunting task. Scalability challenges emerge when attempting to maintain performance across a large number of network nodes and devices.

Implication: Network congestion may become more challenging to manage as the scale of networks increases, potentially leading to congestion collapse.

Limitations of Current Approaches

1. Reaction to Bursty Traffic

Limitation: Current congestion control approaches may struggle to effectively handle bursty traffic patterns, where short periods of intense data transmission are followed by periods of inactivity.

Impact: Inefficient reactions to bursty traffic may result in underutilization of available bandwidth or, conversely, network congestion during peak periods.

2. Fairness in Resource Allocation

Limitation: Ensuring fair resource allocation among different network flows remains a challenge. Existing congestion control mechanisms may not always guarantee equitable distribution of resources, leading to potential disparities among users.

Impact: Unfair resource allocation can result in a suboptimal user experience and may hinder the principles of network neutrality.

3. Real-time Applications

Limitation: Congestion control mechanisms may struggle to meet the demands of real-time applications, such as video streaming and online gaming, where low latency is crucial.

Impact: Suboptimal performance in real-time applications can lead to buffering, latency issues, and a degraded user experience.

Areas for Improvement

1. Machine Learning Integration

Opportunity: Exploring the integration of machine learning algorithms for congestion control offers the potential for adaptive and predictive adjustments based on historical network data.

Benefit: Machine learning models can enhance the ability to predict and react to network conditions, leading to more intelligent and dynamic congestion control.

2. Protocol-Agnostic Approaches

Opportunity: Developing congestion control mechanisms that are agnostic to specific transport protocols could enhance interoperability and simplify integration across diverse networking environments.

Benefit: Protocol-agnostic approaches facilitate seamless communication between systems using different transport protocols.

3. Edge Computing Considerations

Opportunity: Considering the unique challenges of edge computing environments, tailoring congestion control mechanisms to the edge can optimize performance for distributed and edge-centric applications.

Benefit: Edge-specific congestion control can improve the efficiency of data transfer in edge computing scenarios, reducing latency and enhancing overall responsiveness.

4. Collaborative Congestion Control

Opportunity: Exploring collaborative congestion control mechanisms where network nodes actively communicate and coordinate congestion management can lead to more efficient resource utilization.

Benefit: Collaborative approaches foster better coordination among network elements, reducing the likelihood of congestion collapse and improving overall network performance.

In addressing these challenges, overcoming current limitations, and exploring areas for improvement, the field of congestion control can evolve to meet the demands of increasingly complex and dynamic networking environments. These considerations are crucial for enhancing the reliability, efficiency, and fairness of congestion control mechanisms in modern communication networks.

VIII. Conclusion

In conclusion, the study provides valuable insights into congestion control mechanisms in TCP and QUIC. The exploration of QUIC's Cubic algorithm, comparison of loss detection methods, and understanding of protocol-specific approaches contribute to the discourse on network protocol efficiency. Future research should focus on refining the Cubic algorithm, integrating machine learning for adaptability, fostering cross-protocol interoperability, conducting real-world testing, and exploring collaborative congestion control models. These efforts aim to enhance congestion control's adaptability, scalability, and overall performance in evolving networking landscapes.

IX. REFERENCES

[1]. Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

[2]. [FACK] Mathis, M. and J. Mahdavi, "Forward acknowledgement: Refining TCP Congestion Control", ACM SIGCOMM Computer Communication Review, DOI 10.1145/248157.248181, August 1996, <https://doi.org/10.1145/248157.248181>.

[3]. [RETRANSMISSION] Karn, P. and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols", ACM Transactions on Computer Systems, DOI 10.1145/118544.118549, November 1991, <https://doi.org/10.1145/118544.118549>.

[4]. [RFC2018] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, DOI 10.17487/RFC2018, October 1996, <https://www.rfc-editor.org/info/rfc2018>.

[5]. https://storage.googleapis.com/pub-tools-public-publication-

data/pdf/8b935debf13bd176a08326738f5f88ad1 15a071e.pdf

[6]. [RFC8985] Cheng, Y., Cardwell, N., Dukkipati, N., and P. Jha, "The RACK-TLP Loss Detection Algorithm for TCP", RFC 8985, DOI 10.17487/RFC8985, February 2021, <https://www.rfc-editor.org/info/rfc8985>.

Cite this article as :

Vaishnavi Gandhi, Guna Khambhammetu, Ritikaa Kailaas, "From Slow Start to Warp Speed : Congestion Control Tales of QUIC and TCP", International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT), ISSN : 2456-3307, Volume 1, Issue 1, pp.15-23, January-February-2024. Available at doi : https://doi.org/10.32628/CSEIT2390664
Journal URL : https://ijsrcseit.com/CSEIT2390664