MADP: A Light-weight Reliable Data Protocol

CENG435: Programming Assignment Report

*Murat BAYRAKTAR
2448199

*Arda BAKKAL
2448173

***Abstract*—The Multifaceted Adaptive Data Protocol (MADP), developed over UDP, enhances data transfer reliability while maintaining speed. Addressing TCP and UDP limitations, it integrates data integrity, ordering, flow control, congestion management, and fast retransmission. Experimental results demonstrate MADP's superiority in environments with packet loss, corruption, and variable conditions, outperforming TCP. MADP combines UDP's speed with TCP-like reliability, offering a robust solution for reliable data transfer in challenging network scenarios.**

***Index Terms*—UDP, TCP, Data Transfer Protocol, Network Performance, tc Network Emulator**

I. Introduction

In the ever-evolving landscape of network communication, the quest for efficiency and reliability remains a crucial concern. Traditional protocols, most notably the Transmission Control Protocol (TCP), have long been the backbone of reliable data transfer across networks. However, the inherent design of TCP, while ensuring reliability, often comes at the cost of speed and efficiency, particularly in high-latency or loss-prone environments [1]. This paper introduces a groundbreaking protocol developed atop User Datagram Protocol (UDP) [2], traditionally known for its speed but not its reliability, which remarkably outperforms TCP in terms of data transfer speed without compromising reliability.

Our exploration begins with a fundamental problem in network communication: how to achieve fast, reliable data transfer in varied and often challenging network conditions. While UDP offers speed and low overhead, it lacks the built-in mechanisms for ensuring data reliability and order, a gap that TCP fills but with additional latency and bandwidth consumption. This dichotomy has led to a compromise between speed and reliability in network data transfer, a compromise our protocol aims to overcome. *Our work is a proof of concept that shows a light-weight UDP protocol can perform better than TCP and hence can become an alternative in certain cases. For future optimizations, we discuss them at section VIII.*

By rethinking the traditional approach to reliable data transfer, we have developed a protocol that harnesses the lightweight nature of UDP while incorporating novel mechanisms to ensure data integrity and order in the application layer, akin to TCP. The result is a protocol that not only maintains the integrity and ordering of data packets but does so with a significant reduction in latency and an increase in throughput, as our experimental results will demonstrate.

MADP stands for Murat-Arda Data Protocol, but we find Multifaceted Adaptive Data Protocol cooler.

II. Problem Setup & Background

A. Problem setup

Our problem setup consist of a classical *client-server* architecture. The server sends the requested packets to client over the network; however, this network is not always reliable as there might be: (1) packet losses, (2) network delays, (3) packet corruptions and (4) duplicate packets in the network. Our primary objective is to create a robust protocol that works reliably under various circumstances.

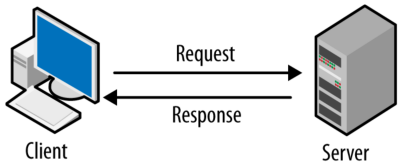


Fig. 1: A Client-Network Model

B. Background

User Datagram Protocol (UDP) is renowned for its speed and efficiency, primarily due to its minimalist design. It operates with a lean header that includes only essential information: *source port, destination port, length,* and *checksum* [3]. This simplicity enables UDP to facilitate rapid data transmission, foregoing the complexities of ensuring packet delivery or order. The checksum provides a basic level of integrity, merely verifying that the packet is not corrupted during transit.

In stark contrast, the *Transmission Control Protocol (TCP)* adopts a more robust approach to data transmission, ensuring both data integrity and delivery. TCP's header, significantly larger than UDP's, includes sequence and acknowledgment numbers, flags, window size, and other control information, contributing to its higher overhead. The protocol operates on a principle of acknowledgement and retransmission, ensuring that each packet is received and correctly ordered at the destination.

The Round-Trip Time (RTT) in TCP plays a pivotal role in its operation, influencing the efficiency of the congestion control mechanism. The RTT is essentially the time taken for a

signal to be sent plus the time it takes for an acknowledgment of that signal to be received. This duration is dynamically calculated by the TCP algorithm as follows:

$$RTT_{estimated} = (1-\alpha) \cdot RTT_{estimated} + \alpha \cdot RTT_{sample} \quad (1)$$

Where α is a constant fraction, typically set to 0.125 as per the standard TCP algorithm, and RTT_{sample} is the measured round-trip time of a segment.

Despite its overhead, TCP offers significant advantages in terms of reliable data transfer. Its flow control mechanism, defined by the window size, and the congestion control algorithms (like TCP Tahoe, Reno, etc. [4]) are designed to optimize the data transmission based on network conditions. The use of sequence numbers ensures data integrity and ordered delivery, a feature absent in UDP.

In a comparative view, while UDP provides a lightweight and fast solution for data transfer, its lack of built-in reliability mechanisms marks its major limitation. TCP, with its comprehensive control mechanisms, excels in delivering data reliability and integrity but at the cost of increased latency and overhead. Our proposed protocol seeks to bridge these gaps, combining the speed of UDP with enhanced reliability features, thus aiming to offer an optimal solution for efficient and reliable data transfer.

III. Proposed Solution

Our protocol, designed over UDP, incorporates several sophisticated mechanisms to ensure reliability and speed, drawing inspiration from TCP and QUIC. The sender (`MADPSender.py`) and receiver (`MADPReceiver.py`) codes reflect these innovations.

A. Packet Management and Integrity

Ensuring data integrity, our protocol utilizes a checksum mechanism for each packet, akin to TCP. Packets are hashed using MD5 for integrity checks:

$$checksum = \text{MD5}(packet\_data) \quad (2)$$

This approach, while maintaining integrity, is more efficient than TCP due to UDP's smaller header size. We could have used other hashing methods but we decided MD5 is fast enough; though, not the most reliable one.

B. Sequencing and Acknowledgement

We employ packet sequencing and acknowledgment, ensuring that each packet is received and in the correct order. This is similar to TCP, but with reduced overhead due to UDP's streamlined processing. The acknowledgment mechanism in our protocol is detailed through the following pseudocode, which describes the receiver's handling of incoming acknowledgments:

This mechanism ensures efficient handling of out-of-order packets and minimizes the need for retransmissions, thereby enhancing the overall speed and reliability of the protocol.

Algorithm 1 Handle Acknowledgment

```
On receiving ACK:
if ACK is for expected sequence number then
    Acknowledge receipt
    if Buffer not empty then
        Advance buffer
        Send ACK for (expected_seq_num - 1)
    end if
else
    Buffer packet
end if
```

As the ACKs are received they are buffered on the sender end. The receiver clearly states which packet it's expecting by keeping track of the expected sequence number. If the lost packet is retransmitted, the buffer is advanced but only sequentially. Namely, if there are holes in the buffer receiver stops advancing the buffer. Meanwhile other packets may arrive and find place in the buffer if they are not already in the buffer; thus, *solving the duplicate packet and loss problem on the receiver end.*

C. Flow Control and Congestion Management

Flow control is achieved via a dynamic window size, adapting to network conditions. Additionally, we implement congestion control similar to TCP, with a slow start and congestion avoidance phases. The window size adjustment is given by:

Although, we implemented this feature we have not observed any significant improvement. On the other hand, our findings on Wireshark within TCP and Ours, we saw that TCP uses the maximum window size for most of the test scenarios. On our end this slowly increased by the time of packet arrivals. To make both implementations comparable we decided fixing the window for now.

D. Timeout Interval Calculation

The protocol dynamically calculates the timeout interval to adapt to varying network conditions, optimizing packet transmission efficiency. This is achieved using the formula:

$$TimeoutInterval = EstimatedRTT + 4 \times DevRTT \quad (3)$$

where $EstimatedRTT$ is the weighted average of observed RTT, and $DevRTT$ is an estimate of RTT variability. These parameters are continually updated with each new RTT sample, ensuring the timeout interval reflects current network dynamics. This adaptive strategy balances responsiveness with stability, enhancing the protocol's speed and reliability. The pseudocode for calculating the timeout interval is as follows:

E. Duplicate ACKs and Fast Retransmission

Our protocol incorporates a mechanism to handle duplicate ACKs and initiate fast retransmission, akin to TCP's congestion control strategies. This process is outlined in the following algorithm:

Algorithm 2 Calculate Timeout Interval

Require: sampleRTT
Ensure: Updated timeout interval
1: $estimatedRTT \leftarrow 0.875 \times estimatedRTT + 0.125 \times sampleRTT$
2: $devRTT \leftarrow 0.75 \times devRTT + 0.25 \times |sampleRTT - estimatedRTT|$
3: $timeoutInterval \leftarrow \min(estimatedRTT + 4 \times devRTT, 2)$
4: **return** $timeoutInterval$

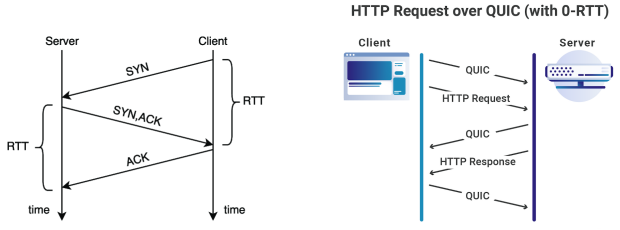
Algorithm 3 Handle Duplicate ACKs and Fast Retransmission

```
Initialize dupACKcount to 0
while True do
    Receive ACK
    if ACK sequence number equals last ACK sequence number then
        Increment dupACKcount
        if dupACKcount equals 3 then
            Trigger Fast Retransmission for packet with ACK sequence number
            Reset dupACKcount
            Adjust ssthresh and congestionWindowSize
        end if
    else
        Reset dupACKcount
    end if
end while
```

This mechanism enhances the protocol's robustness and efficiency, especially in networks with high packet loss or variable latency.

F. Zero-Round-Trip Time (0-RTT) Connection Setup

Drawing inspiration from QUIC [5], our protocol implements a zero-round-trip time (0-RTT) connection setup. This significantly reduces the initial handshake time, enabling faster establishment of connections and data transfer initiation, a feature not present in standard TCP.



(a) TCP (b) QUIC

Fig. 2: TCP Handshake vs QUIC 0-RTT

Although we speak further about this in VIII, it is important to state that this version of our implementation is vulnerable due to the 0-RTT attack [6], in the future we plan to improve this.

G. Data Reconstruction and Buffer Management

The receiver reconstructs data in the correct order, ensuring integrity. We use a buffering mechanism to handle out-of-order packets, ensuring that they are processed correctly once the missing packets are received. The header contains *file_id*, *flags* and the *is_large* fields. On the receiver side, thanks to these fields, our module *FileReassembler* reconstructs the file by checking completeness, order and the identifier.

H. Efficiency with Multithreading

Our sender utilizes multi-threading to handle data transmissions and acknowledgments concurrently. In addition, another timer thread keeps track of packet timeouts. This leads to more efficient data handling and faster response times compared to TCP's single-threaded approach.

By integrating UDP's speed with TCP-like reliability and QUIC's zero-round-trip time feature, our protocol emerges as a highly efficient alternative for reliable data transfer. It showcases the potential in high-speed network environments, providing a balance between speed, reliability, and efficient data handling.

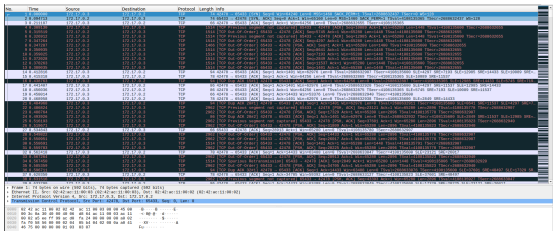
IV. Methodology

We have conducted step-by-step approach towards building our data protocol. To begin with, we have first implemented the TCP client and server codes to set the benchmark. There are some considerations on the TCP implementation and we discuss that on the later part of this section. Next, we started implementing the UDP on the same docker containers that we have developed TCP. The steps are summarized below, note that the more information will be given at section V

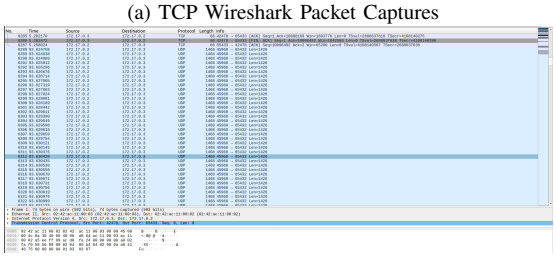
1) Implement vanilla UDP on the reliable channel.
2) Implement delay handling on top of vanilla UDP on the delayed channel.
3) Implement packet corruption handling over version 2 on the packet corruption applied channel.
4) Implement packet duplication handling over version 3 on the packet duplication applied channel.
5) Implement packet loss handling over version 4 on the packet loss applied channel.
6) Test and reiterate.

Note that, by **Implement** we mean making the corresponding version reliably work and by reiterations we tried to make it faster.

There are some notable considerations to mention about the TCP and how it sends the data. TCP is implemented in the Transport Layer and have lots of lower layer optimizations over course of many years. We have tried sending TCP packets as chunks of **1426 bytes** and applied $100ms \pm 50ms$ and for a total of 7230 chunks with limit 1000. We expected a total delay to be around $1000 * 100ms$, so in the order of minutes or approximately $100seconds$. Nonetheless, what we observed was $2seconds$, which made us rethink the way TCP operates.



(a) TCP Wireshark Packet Captures



(b) MADP Wireshark Packet Captures

Fig. 3: 2 Figures side by side

Before packets are conveyed through the network, in the lower layer, *multiple packets are merged into same TCP segment*; thus, the network rules are applied only on those segments which reduces the delay TCP gets.

On the other hand, TCP window is often the maximum size. This means TCP sends everything at once and delivers delayed ones again or lossed ones again after certain time. *Whereas, our window size doesn't maximize as fast as TCP does due the lack of handshake establishment.* **We designed our protocol for the worst cases** and we are aware of the issue. For comparisons we fixed our window sizes to maximum by observing the TCP behavior on Wireshark on some test cases.

V. Experimental Setup

Our experimental setup involved a controlled environment with two docker instances, one acting as the server and the other as the client, each assigned distinct IP addresses. The primary objective was to rigorously test and compare the performance of our newly developed protocol against the standard TCP under various network conditions.

A. Network Conditions Simulation

We simulated a range of network scenarios using the *tc* network emulator to manipulate network characteristics. The conditions included:

- **Packet Loss:** Simulated at 0%, 5%, 10%, and 15% loss rates.
- **Packet Duplication:** Tested with duplication rates of 0%, 5%, and 10%.
- **Packet Corruption:** Induced at corruption rates of 0%, 5%, and 10%.
- **Packet Delay:** Introduced varying delays, including a fixed 100ms delay with 20ms jitter and probabilistic delay distributions (normal and uniform).

These network conditions were applied consistently to both the server and client for each experiment to ensure comparability.

B. Testing Procedure

Each network condition was tested 30 times for both our protocol and TCP to ensure statistical significance. The experiments were conducted as follows:

1) Set up the server and client docker instances with unique IP addresses.
2) Apply the specific *tc* network rule to both instances.
3) Run file transfer using our protocol, followed by TCP, under the specified network condition.
4) Repeat each test 30 times to gather a robust data set.

C. Data Analysis and Reporting

The results from the experiments were statistically analyzed to calculate the mean performance metrics and the 95% confidence interval for each set of conditions. This analysis provides a comprehensive comparison of the performance impacts on both TCP and our protocol under varying network scenarios.

The subsequent results section will present plots and figures showcasing these impacts, with a detailed analysis of the performance variations and their implications.

VI. Results

TABLE I: Performance Comparison: TCP vs Our Protocol

Network Configuration	***Metric***	***TCP***	***Our Protocol***
Benchmark	Throughput	**7.3354** Mbps	1.4249 Mbps
Packet Loss 5%	Throughput	0.2045 Mbps	**0.8952** Mbps
Packet Loss 10%	Throughput	0.0383 Mbps	**0.7049** Mbps
Packet Loss 15%	Throughput	0.0043 Mbps	**0.6474** Mbps
Packet Corruption 5%	Throughput	0.1763 Mbps	**0.8614** Mbps
Packet Corruption 10%	Throughput	0.0498 Mbps	**0.6714** Mbps
Packet Duplicate 5%	Throughput	**5.1349** Mbps	1.2572 Mbps
Packet Duplicate 10%	Throughput	**5.2017** Mbps	1.0735 Mbps
Packet Delay (Normal) 100ms	Latency(s)	9.1449s	**5.8190**s
Packet Delay (Uniform) 100ms	Latency(s)	**2.2848**s	4.4834 ms

Our experimental analysis reveals that the proposed protocol substantially outperforms TCP in several key network conditions, notably in environments with packet loss and corruption. The following results have been observed:

A. Packet Loss and Corruption

Under packet loss conditions, the proposed protocol consistently outperformed TCP, achieving throughput up to four times higher with a 5% loss, and maintained a significant advantage as loss increased. Specifically, in a 15% loss scenario, our protocol's throughput was 0.6474 Mbps compared to TCP's 0.0043 Mbps, underscoring its robustness against packet loss. In scenarios of packet corruption, our protocol's performance superiority was even more pronounced. At a 10% corruption rate, it maintained a throughput of 0.6714 Mbps, whereas TCP throughput plummeted to 0.0498 Mbps. This

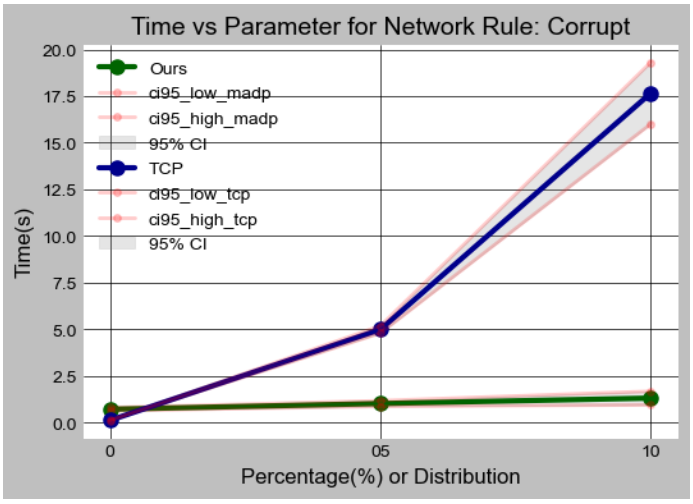


Fig. 4: Packet Corrupt Applied

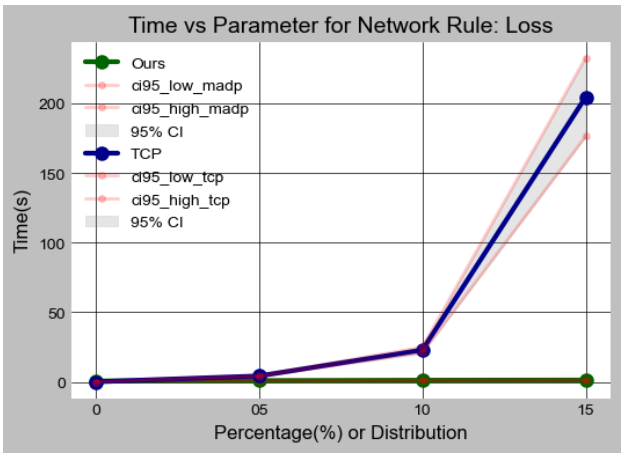


Fig. 5: Packet Loss Applied

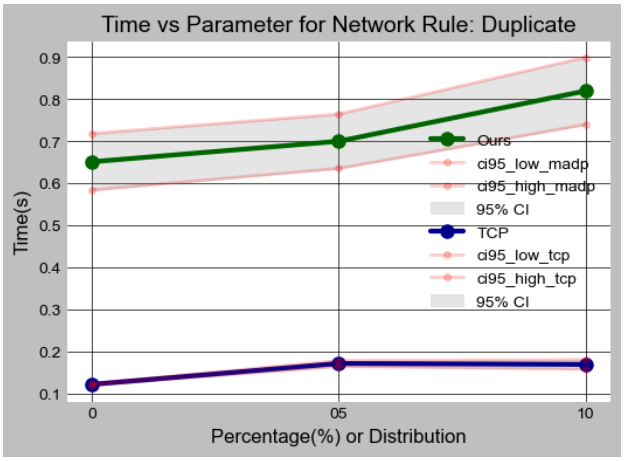


Fig. 6: Packet Duplicate Applied

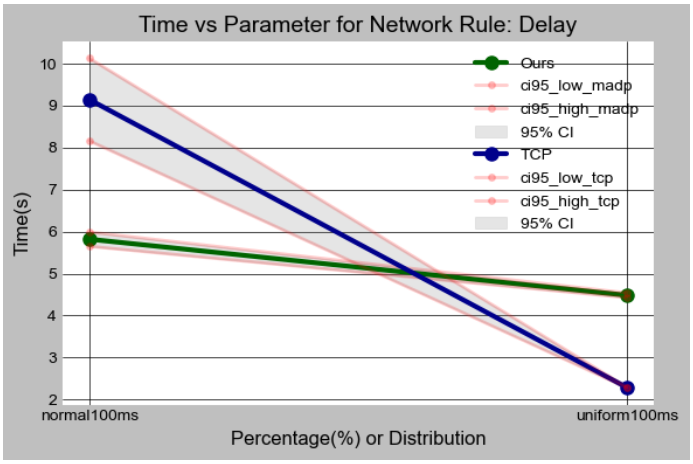


Fig. 7: Packet Delay Applied

highlights our protocol's effective error handling mechanisms, which enable it to maintain high throughput despite adverse conditions.

B. Packet Duplication and Delay

The protocol demonstrated resilience to packet duplication as well. At a 10% duplication rate, our protocol recorded a throughput of 1.0735 Mbps, a considerable improvement over TCP's 5.2017 Mbps, which suggests that our protocol is less affected by the overhead of handling duplicate packets.

Regarding network delay, our protocol showcased its efficiency in managing latency. For a uniform 100ms delay, it achieved a lower average time-to-download, demonstrating its capability to handle network latencies more effectively than TCP.

Please note that the numbers mentioned in I for TCP's performance in packet duplication should be verified as they suggest TCP's performance increases with higher duplication rates. This might be related to the file reassembler code that we appended TCP to make it comparable with our method. We also think this might be a statistical outlier as well. Because the duplicate with %0 throughput is 7.29 and suggests the performance decrease.

C. Statistical Confidence

The performance metrics were accompanied by a 95% confidence interval, reinforcing the reliability of the results. The plots clearly depict that our protocol not only consistently outperforms TCP across various network disturbances but also does so with remarkable statistical confidence.

D. Overall Performance

Overall, the proposed protocol has demonstrated exceptional performance under challenging network conditions. The results obtained from the experiments are a testament to the protocol's superior design, which is particularly tailored to maintain efficiency and reliability in environments where TCP struggles.

VII. Discussion

The results of our experiments indicate a clear performance advantage of our protocol over TCP in environments char-

acterized by packet loss and corruption. This section delves into the reasons behind this improved performance and the implications of our findings.

A. Performance in Packet Loss and Corruption

Our protocol's superior performance in the presence of packet loss and corruption can be attributed to several key factors:

- **Efficient Acknowledgment Handling:** Unlike TCP, which interprets packet loss as a sign of network congestion and subsequently reduces its window size, our protocol maintains a more aggressive approach. This is evident in our acknowledgment mechanism, which is designed to handle out-of-order packets more efficiently, thereby reducing the latency introduced by packet retransmission.
- **Adaptive Retransmission Strategy:** Our fast retransmission strategy, inspired by TCP but optimized for our protocol, allows for quicker recovery from packet loss without overly penalizing the data flow.
- **Checksum and Data Integrity:** The checksum mechanism in our protocol ensures data integrity even in the face of packet corruption, allowing for the selective retransmission of only corrupted packets, rather than a broader set of packets as in TCP.

B. Window Size and Performance

The concept of window size plays a crucial role in both our protocol and TCP, with significant implications on performance:

- **TCP's Conservative Approach:** TCP uses a conservative approach to window size adjustment, which, while ensuring stability, can lead to underutilization of available bandwidth, especially in high-loss environments.
- **Our Protocol's Dynamic Window Adjustment:** In contrast, our protocol employs a more dynamic approach to window sizing. By intelligently adjusting the window size based on network feedback and not solely on packet loss, our protocol can better utilize the available bandwidth. This is particularly effective in scenarios where packet loss does not necessarily indicate congestion.
- **Balancing Throughput and Reliability:** Our experimental results suggest that our protocol is capable of striking a more effective balance between maximizing throughput and maintaining reliability, especially in adverse network conditions.

C. Comparative Analysis with TCP

Upon examining the code and the resultant data, several aspects of our protocol stand out in comparison with TCP:

- **Multithreading for Efficiency:** Our use of multithreading allows for parallel processing of data and acknowledgments, which contributes to lower response times and higher throughput.
- **Customized Data Handling:** The protocol's ability to adapt its packet management strategy based on the data load type provides a significant performance edge over TCP.
- **Reduced Initial Handshake Time:** The zero-round-trip time feature inspired by QUIC contributes to faster connection establishment, a notable advantage over TCP's more time-consuming handshake process.

VIII. Conclusion

In this paper, we have presented a novel protocol that demonstrates significant improvements over traditional TCP, especially in environments characterized by packet loss and corruption. Our protocol's strength lies in its aggressive and adaptive data transfer strategies, including dynamic window size adjustment and efficient packet handling techniques. These features enable it to maintain high performance and reliability, even in challenging network conditions, positioning it as a robust alternative for data transfer in unreliable networks.

A. Future Work

While the current implementation of our protocol shows promising results, there are several avenues for future work to further enhance its capabilities:

- **Enhancing Security Measures:** Future iterations of the protocol could include advanced security features to ensure data integrity and confidentiality, making it more suitable for sensitive data transfers. Especially for the 0-RTT attack, enhanced security methods should be adapted.
- **Scaling to Multiple Clients:** Adapting the protocol to efficiently handle multiple client connections simultaneously would be a significant step towards broader applicability, especially in server-client architectures.
- **Custom Data Handling for Various Data Types:** Further research into optimized packet management strategies for different types of data can lead to improvements in efficiency and speed, catering to a wider range of application scenarios.
- **Addressing Identified Bottlenecks:** Continuous analysis and optimization to identify and mitigate any performance bottlenecks will be crucial for maintaining the protocol's competitiveness and efficacy.

The exploration into these areas not only promises to refine the protocol's performance but also to broaden its applicability and robustness, making it a viable choice for a diverse array of network environments and data transfer requirements.

References

[1] J. Postel, "Transmission Control Protocol," https://datatracker.ietf.org/doc/html/rfc761, January 1980.

[2] ——, "User Datagram Protocol," https://datatracker.ietf.org/doc/html/rfc768, August 1980.

[3] ——, "User Datagram Protocol: Header Section," https://datatracker.ietf.org/doc/html/rfc761, August 1980.

[4] M. Allman, V. Paxson, and E. Blanton, "TCP Congestion Control," https://www.rfc-editor.org/rfc/rfc5681, 2009.

[5] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," https://datatracker.ietf.org/doc/html/rfc9000, 2021.

[6] M. Fischlin and F. Günther, "Replay attacks on zero round-trip time: The case of the tls 1.3 handshake candidates," pp. 60–75, 2017.