Implementation and Optimization of the Ring Token Protocol

Authors:

Sanchi Sujith Kumar (Roll No: 210010047),

Sree Naga Manikanta Katta (Roll No: 210010050),

Revanth Palaparthi (Roll No: 210010043),

4th year Students, Department of Computer Science and Engineering,

Indian Institute of Technology, Dharwad

Internship at Defence Research and Development Laboratory (DRDL), Hyderabad

Under the guidance of

Mr. Deepanshu Dixit
Scientist-C

May – June 2024

Abstract

This report presents the implementation and optimization of the Ring Token Protocol, a method used for network communication in token ring networks. We aimed to reduce the total time taken for data transfer per packet by streaming multiple packets together. Our results demonstrate a significant improvement in efficiency, suggesting that this optimization technique can effectively enhance the performance of token ring networks.

Table of Contents

1. Introduction
2. Methodology
3. Code Snippets
4. Results
5. Discussion
6. Conclusion

1. Introduction

Background

The ring token protocol is a well-established method used in token ring networks to manage the transmission of data packets. In this protocol, a token circulates around the network with nodes interconnected in a ring topology, granting the holder permission to send data and enabling token-based data transmission. This ensures orderly and collision-free communication among nodes.

Problem Statement

Despite its advantages, the Ring Token Protocol often suffers from inefficiencies due to idle times when nodes wait for the token to arrive. These idle times can significantly increase the total data transfer time, especially under high traffic conditions. Therefore, there is a need for optimization.

Objective

This project aims to optimize the ring token protocol by allowing multiple packets, destined for the same node, to stream together in groups, thereby reducing idle times and improving overall data transfer efficiency.

Scope

This report covers the implementation of the ring token protocol, the optimization technique, and the performance evaluation of the optimized protocol. The scope is limited to a simulated network environment with 5 terminals on a single PC.

2. Methodology

Implementation Details

The Ring Token Protocol was implemented using Python. The simulation environment consisted of a network of nodes. Each node, implemented through a terminal or command-line interface and connected via TCP sockets to other nodes, could send and receive packets according to the Ring Token Protocol rules.

Optimization Technique

For optimization, instead of sending one packet at a time, multiple packets destined for the same node are sent together. The sender node sends the token to the receiver node to notify it of the arriving packets, and then the packets are transmitted to the receiver node in a batch.

Why This Technique

This technique was chosen for its capability to reduce node idle times and improve Round-Trip Time (RTT) for a packet. By sending multiple packets destined for the same node together in a batch, the receiving node spends less time waiting for the token to arrive between each packet transmission, thereby reducing its idle time. Additionally, since multiple packets are transmitted in a single batch, the RTT for each packet is reduced as they are delivered more quickly to the destination node. Consequently, this approach results in overall faster data transfer and improved network efficiency, minimizing idle time and enhancing throughput.

Implementation Steps

1. The sender initiates the process by sending the token to the receiver node. The token contains information about the receiver node's identity and the number of packets that are about to be transmitted.
2. Following the token transmission, the sender proceeds to send the packets. Upon reception, the receiver node reads the data contained within each packet.
3. Upon receiving the packets, the receiver node sends an acknowledgment (ACK) along with the token back to the sender.
4. If the sender receives the ACK, it prepares to send packets to the next receiver node. However, if there is no acknowledgment received, indicating that some packets were not received or acknowledged, the

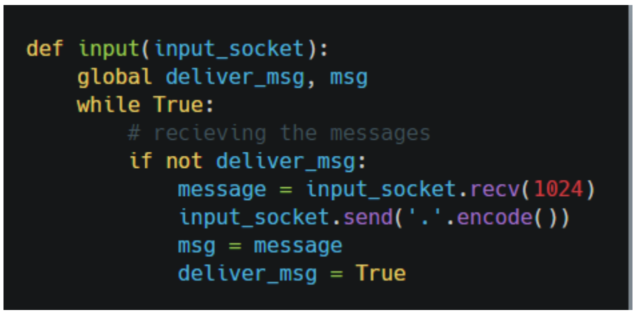
sender retransmits the unreceived packets to the involved receiver node.

5. In the optimized protocol, a constant variable was introduced to define the maximum number of packets that the sender can transmit to a receiver in a single batch. This variable acts as a threshold to control the number of packets grouped together for transmission, thereby ensuring that the network remains efficient and does not become overloaded with excessively large batches.
6. Additionally, a node is not allowed to hold the token beyond a predetermined threshold value. This threshold ensures that no single node monopolizes the token, which could otherwise lead to inefficiencies and increased latency for other nodes waiting to transmit their data. By enforcing this limit, the protocol maintains a balanced and fair distribution of network resources, preventing delays and promoting smooth data flow across the network.
7. In the optimized protocol, the sender follows a round-robin format while transmitting packets to the receiver nodes. This means that if the sender sends a batch of packets to node X, it will not send any additional packets to node X until it has sent packets or a batch to the other nodes in the network. This approach ensures a fair distribution of network resources and prevents any single node from dominating the communication channel, thereby promoting balanced and efficient data transmission.

3. Code Snippets

base_station.py

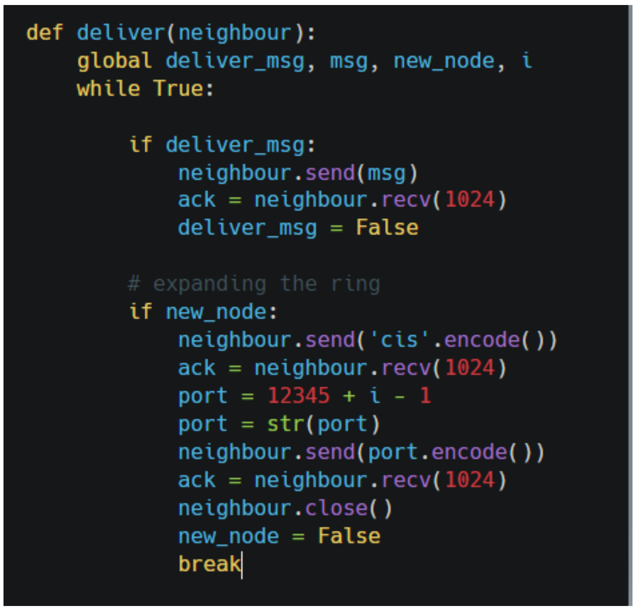
This file will be run first and kept undisturbed. It accepts new connections and ensures they are integrated into the ring, allowing it to expand. The connection is managed in a decentralized manner as per the protocol. Additionally, it sends the token to the first connected node.



```
def input(input_socket):
    global deliver_msg, msg
    while True:
        # recieving the messages
        if not deliver_msg:
            message = input_socket.recv(1024)
            input_socket.send('.'.encode())
            msg = message
            deliver_msg = True
```

Figure 1: input() function

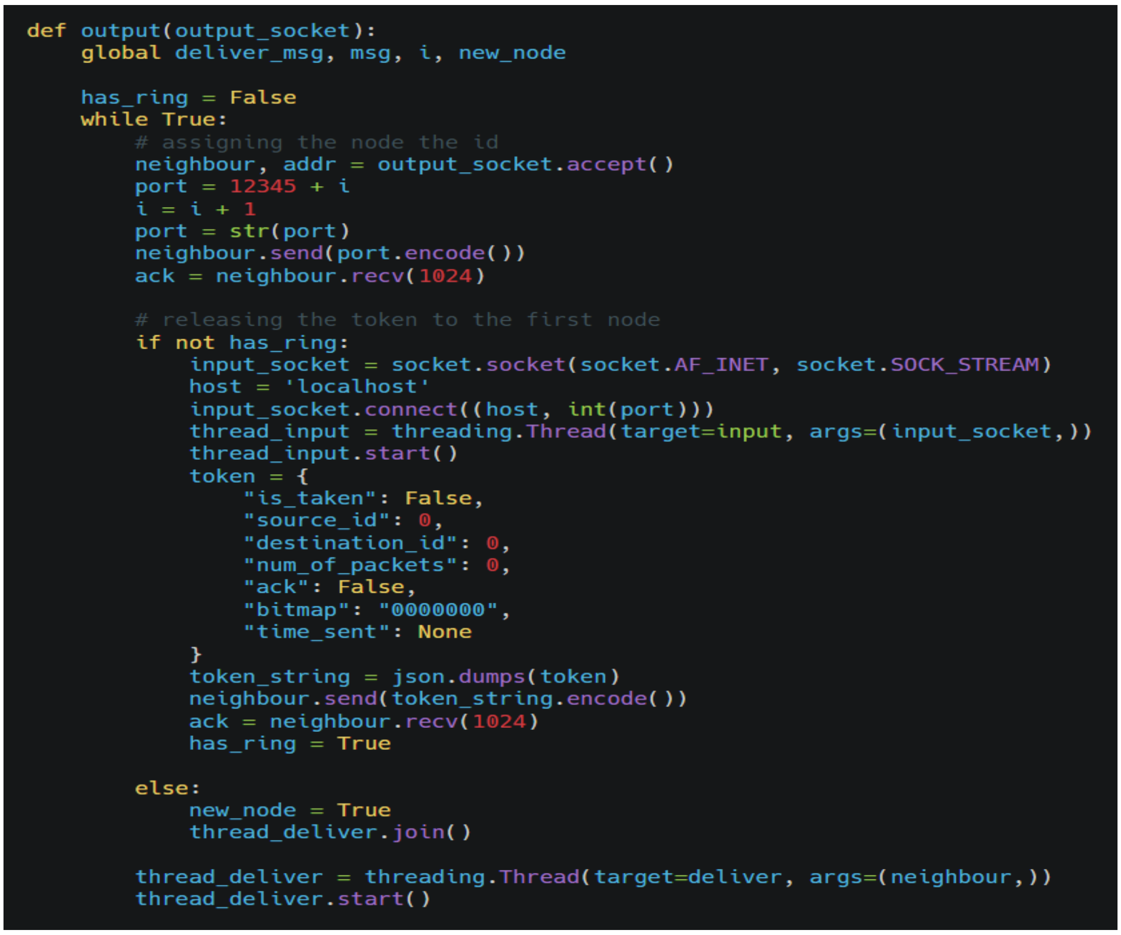
The **input()** function just delivers the message to next node.



```python
def deliver(neighbour):
    global deliver_msg, msg, new_node, i
    while True:

        if deliver_msg:
            neighbour.send(msg)
            ack = neighbour.recv(1024)
            deliver_msg = False

        # expanding the ring
        if new_node:
            neighbour.send('cis'.encode())
            ack = neighbour.recv(1024)
            port = 12345 + i - 1
            port = str(port)
            neighbour.send(port.encode())
            ack = neighbour.recv(1024)
            neighbour.close()
            new_node = False
            break
```

Figure 2: deliver() function

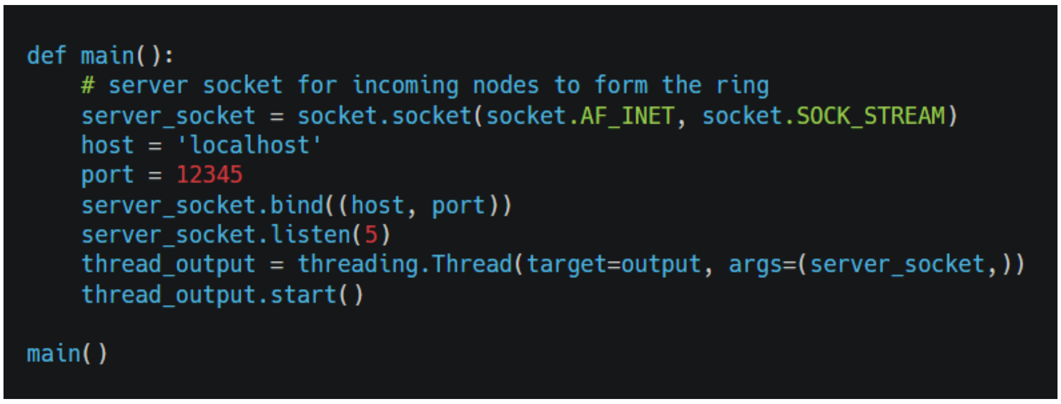
The **deliver()** function sends the port number to the newly connected node whenever a new connection is established.



```python
def output(output_socket):
    global deliver_msg, msg, i, new_node

    has_ring = False
    while True:
        # assigning the node the id
        neighbour, addr = output_socket.accept()
        port = 12345 + i
        i = i + 1
        port = str(port)
        neighbour.send(port.encode())
        ack = neighbour.recv(1024)

        # releasing the token to the first node
        if not has_ring:
            input_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            host = 'localhost'
            input_socket.connect((host, int(port)))
            thread_input = threading.Thread(target=input, args=(input_socket,))
            thread_input.start()
            token = {
                "is_taken": False,
                "source_id": 0,
                "destination_id": 0,
                "num_of_packets": 0,
                "ack": False,
                "bitmap": "0000000",
                "time_sent": None
            }
            token_string = json.dumps(token)
            neighbour.send(token_string.encode())
            ack = neighbour.recv(1024)
            has_ring = True

        else:
            new_node = True
            thread_deliver.join()

        thread_deliver = threading.Thread(target=deliver, args=(neighbour,))
        thread_deliver.start()
```

Figure 3: output() function

The **output()** function expands the ring through new connections and in case of the first node connection, it also releases the token to that node**.**

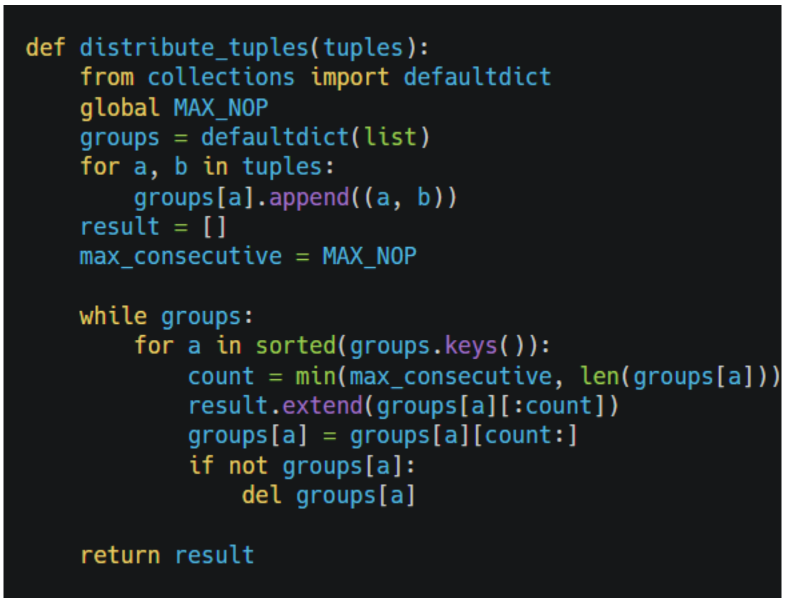


```python
def main():
    # server socket for incoming nodes to form the ring
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    host = 'localhost'
    port = 12345
    server_socket.bind((host, port))
    server_socket.listen(5)
    thread_output = threading.Thread(target=output, args=(server_socket,))
    thread_output.start()

main()
```

Figure 4: main() function

node.py

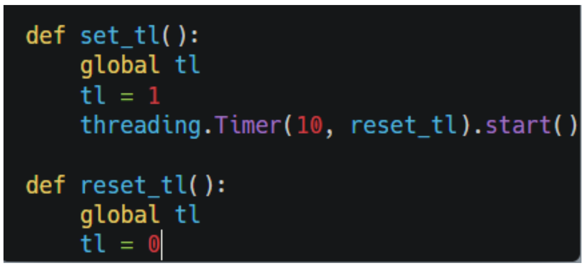
Each node is implemented through a terminal/command-line interface, facilitating protocol execution. Upon execution of this file, the node establishes a connection with the base station, which assigns it a port and ID. Through this initial connection, the ring network connections are established. The node is equipped with an input socket for sending tokens and data, as well as an output socket for receiving them.



```python
def distribute_tuples(tuples):
    from collections import defaultdict
    global MAX_NOP
    groups = defaultdict(list)
    for a, b in tuples:
        groups[a].append((a, b))
    result = []
    max_consecutive = MAX_NOP

    while groups:
        for a in sorted(groups.keys()):
            count = min(max_consecutive, len(groups[a]))
            result.extend(groups[a][:count])
            groups[a] = groups[a][count:]
            if not groups[a]:
                del groups[a]

    return result
```

Figure 5: distribute_tuples() function

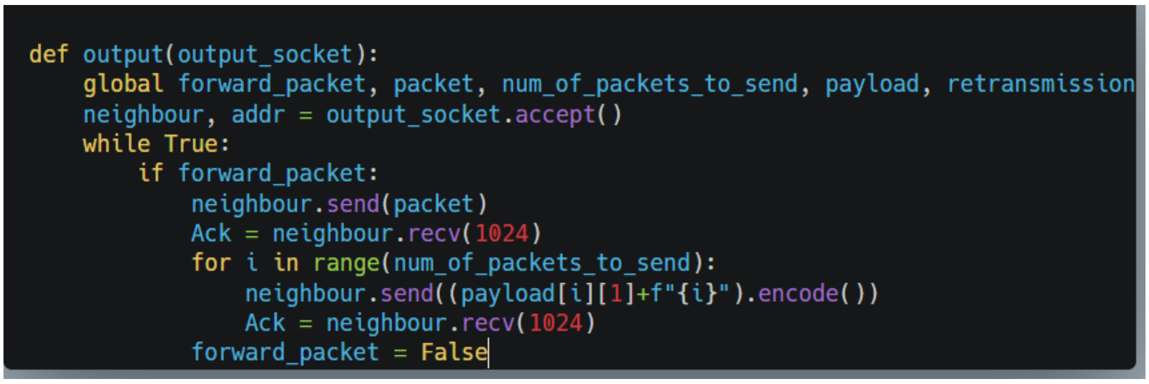
The **distribute_tuples()** function distributes the tuples evenly to ensure a round-robin type format while delivering messages.



```python
def set_tl():
    global tl
    tl = 1
    threading.Timer(10, reset_tl).start()

def reset_tl():
    global tl
    tl = 0
```

Figure 6: set_tl() and reset_tl() functions

The **set_tl()** and **reset_tl()** functions ensure fairness in token handling, so that each token gets an equal share of the token. It ensures that once a node releases the token, it wont be able to reaquire it until some time has passed.

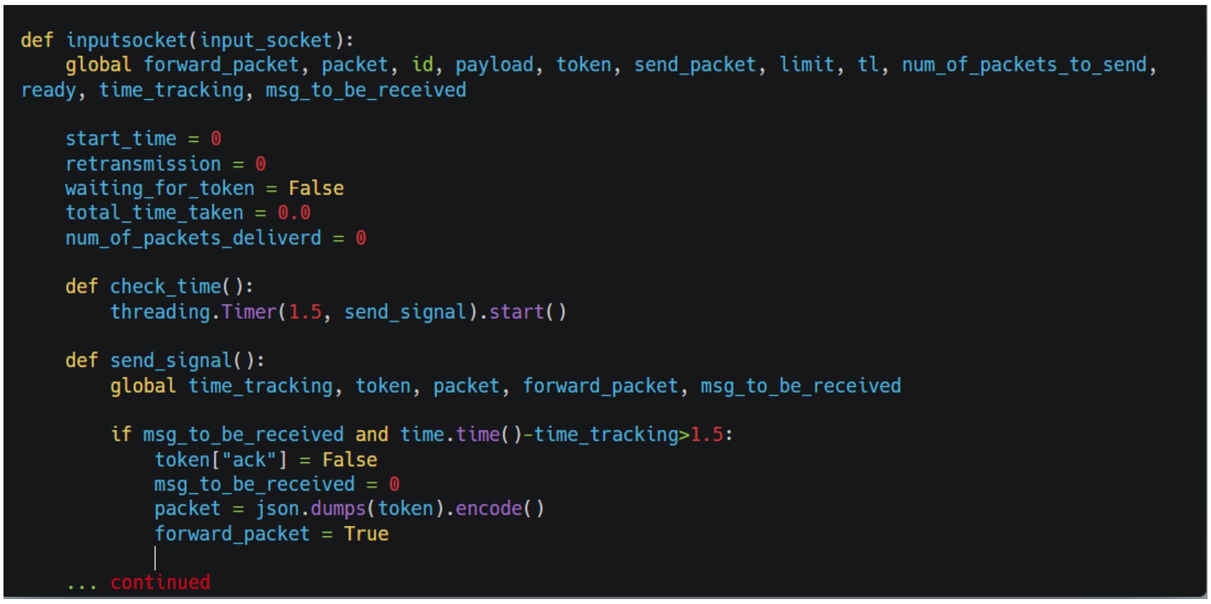


```python
def output(output_socket):
    global forward_packet, packet, num_of_packets_to_send, payload, retransmission
    neighbour, addr = output_socket.accept()
    while True:
        if forward_packet:
            neighbour.send(packet)
            Ack = neighbour.recv(1024)
            for i in range(num_of_packets_to_send):
                neighbour.send((payload[i][1]+f"{i}").encode())
                Ack = neighbour.recv(1024)
            forward_packet = False
```

Figure 7: output() function

The **output()** function just delivers the token and packets(if any) to the neighbouring node.

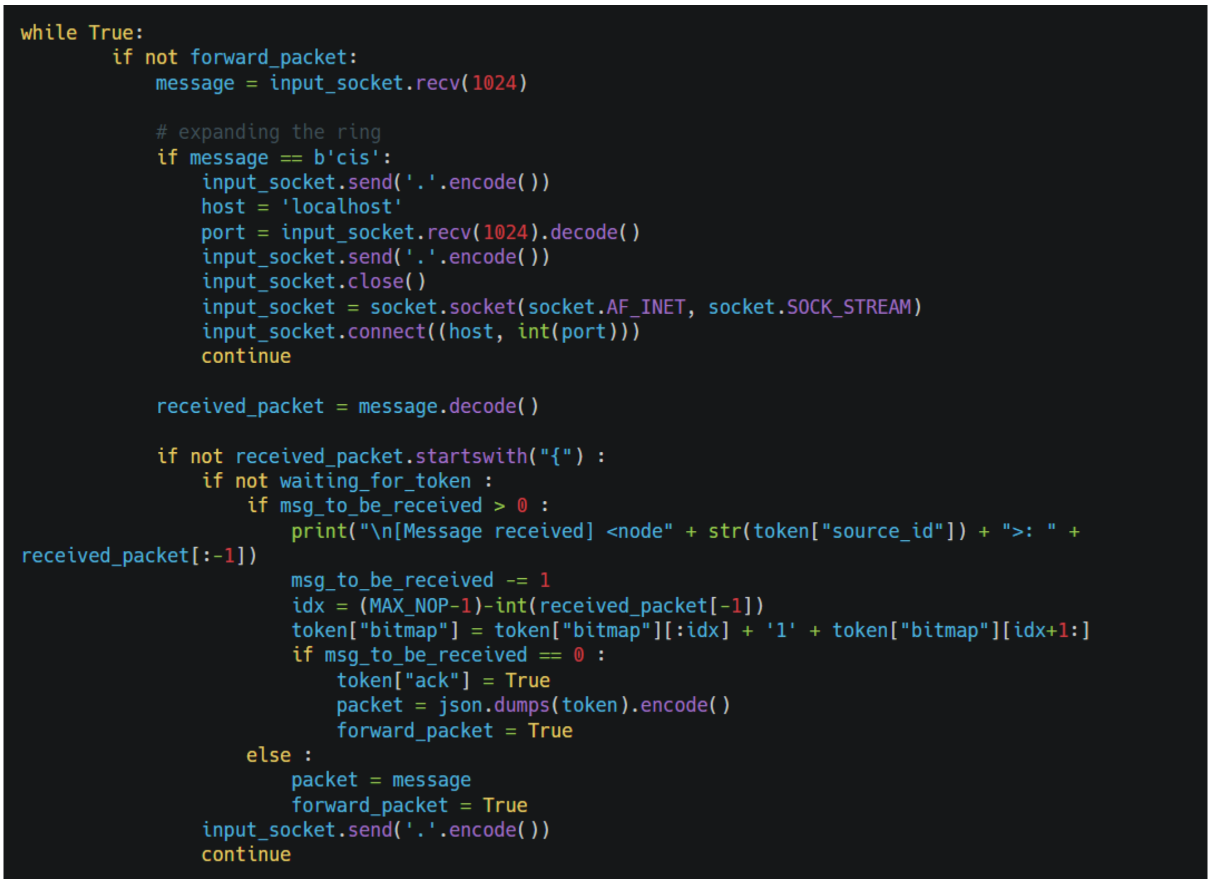
All snippets below are part of the **inputsocket()** function.



```python
def inputsocket(input_socket):
    global forward_packet, packet, id, payload, token, send_packet, limit, tl, num_of_packets_to_send,
ready, time_tracking, msg_to_be_received

    start_time = 0
    retransmission = 0
    waiting_for_token = False
    total_time_taken = 0.0
    num_of_packets_deliverd = 0

    def check_time():
        threading.Timer(1.5, send_signal).start()

    def send_signal():
        global time_tracking, token, packet, forward_packet, msg_to_be_received

        if msg_to_be_received and time.time()-time_tracking>1.5:
            token["ack"] = False
            msg_to_be_received = 0
            packet = json.dumps(token).encode()
            forward_packet = True

    ... continued
```

Figure 8: input_socket() function part-1

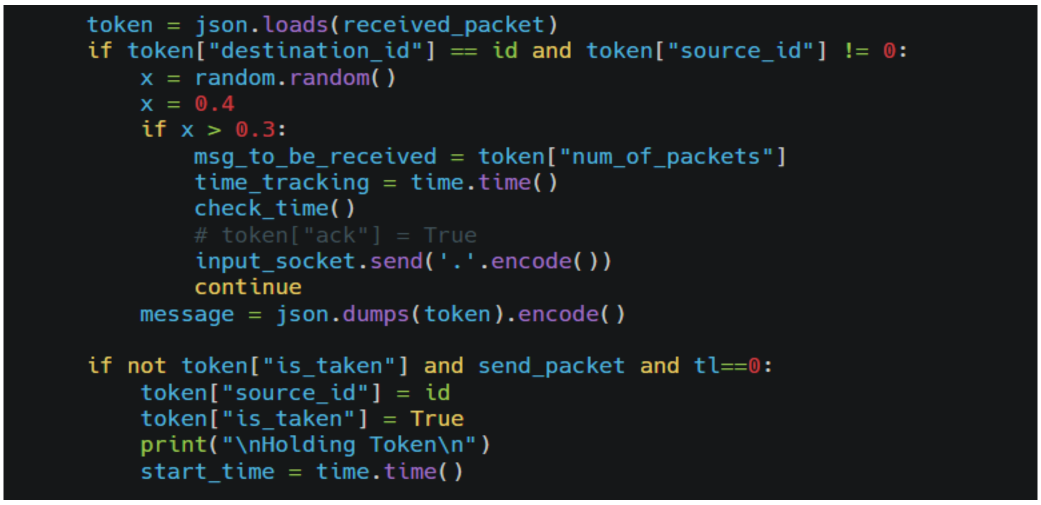
The **check_time()** and **send_signal()** functions send the token back to the sender without acknowledgment if the packets have not arrived within the specified waiting time.



```
while True:
        if not forward_packet:
            message = input_socket.recv(1024)

            # expanding the ring
            if message == b'cis':
                input_socket.send('.'.encode())
                host = 'localhost'
                port = input_socket.recv(1024).decode()
                input_socket.send('.'.encode())
                input_socket.close()
                input_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                input_socket.connect((host, int(port)))
                continue

            received_packet = message.decode()

            if not received_packet.startswith("{") :
                if not waiting_for_token :
                    if msg_to_be_received > 0 :
                        print("\n[Message received] <node" + str(token["source_id"]) + ">: " +
received_packet[:-1])
                        msg_to_be_received -= 1
                        idx = (MAX_NOP-1)-int(received_packet[-1])
                        token["bitmap"] = token["bitmap"][:idx] + '1' + token["bitmap"][idx+1:]
                        if msg_to_be_received == 0 :
                            token["ack"] = True
                            packet = json.dumps(token).encode()
                            forward_packet = True
                    else :
                        packet = message
                        forward_packet = True
                input_socket.send('.'.encode())
                continue
```

Figure 9: input_socket() function part-2

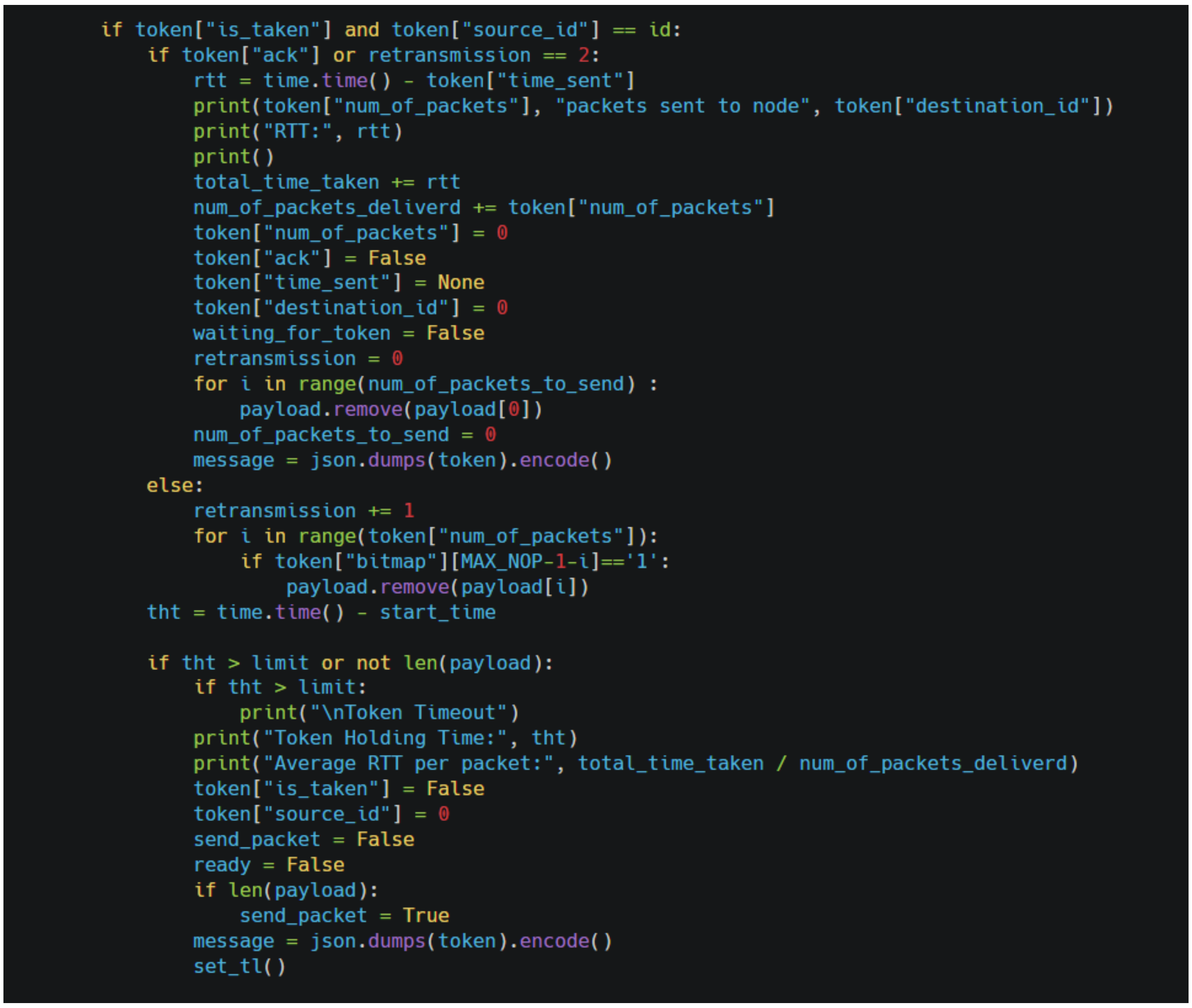
The first ‘if’ block helps in expanding the ring by receiving the port number and establishing connections.
The second ‘if’ block handles the packets received.



```
    token = json.loads(received_packet)
    if token["destination_id"] == id and token["source_id"] != 0:
        x = random.random()
        x = 0.4
        if x > 0.3:
            msg_to_be_received = token["num_of_packets"]
            time_tracking = time.time()
            check_time()
            # token["ack"] = True
            input_socket.send('.'.encode())
            continue
        message = json.dumps(token).encode()

    if not token["is_taken"] and send_packet and tl==0:
        token["source_id"] = id
        token["is_taken"] = True
        print("\nHolding Token\n")
        start_time = time.time()
```

Figure 10: input_socket() function part-3

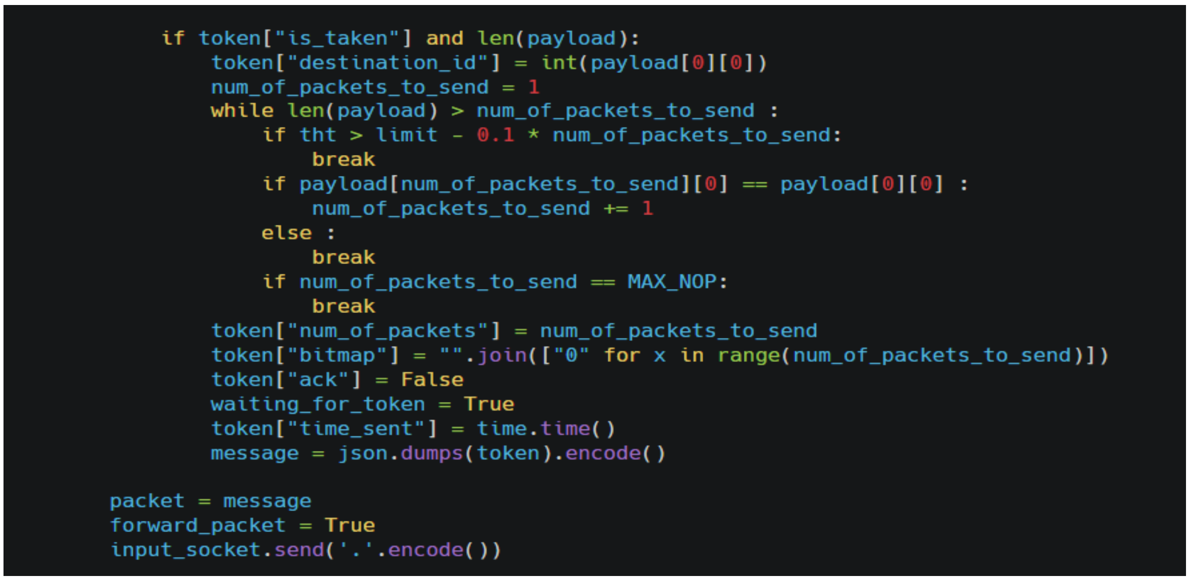
Here, the 1st ‘if’ block handles the case when the token reaches the intended destination node.
2nd ‘if’ block helps in acquiring the token if it is not held by any other node.



```
        if token["is_taken"] and token["source_id"] == id:
            if token["ack"] or retransmission == 2:
                rtt = time.time() - token["time_sent"]
                print(token["num_of_packets"], "packets sent to node", token["destination_id"])
                print("RTT:", rtt)
                print()
                total_time_taken += rtt
                num_of_packets_deliverd += token["num_of_packets"]
                token["num_of_packets"] = 0
                token["ack"] = False
                token["time_sent"] = None
                token["destination_id"] = 0
                waiting_for_token = False
                retransmission = 0
                for i in range(num_of_packets_to_send) :
                    payload.remove(payload[0])
                num_of_packets_to_send = 0
                message = json.dumps(token).encode()
            else:
                retransmission += 1
                for i in range(token["num_of_packets"]):
                    if token["bitmap"][MAX_NOP-1-i]=='1':
                        payload.remove(payload[i])
            tht = time.time() - start_time

            if tht > limit or not len(payload):
                if tht > limit:
                    print("\nToken Timeout")
                print("Token Holding Time:", tht)
                print("Average RTT per packet:", total_time_taken / num_of_packets_deliverd)
                token["is_taken"] = False
                token["source_id"] = 0
                send_packet = False
                ready = False
                if len(payload):
                    send_packet = True
                message = json.dumps(token).encode()
                set_tl()
```

Figure 11: input_socket() function part-4

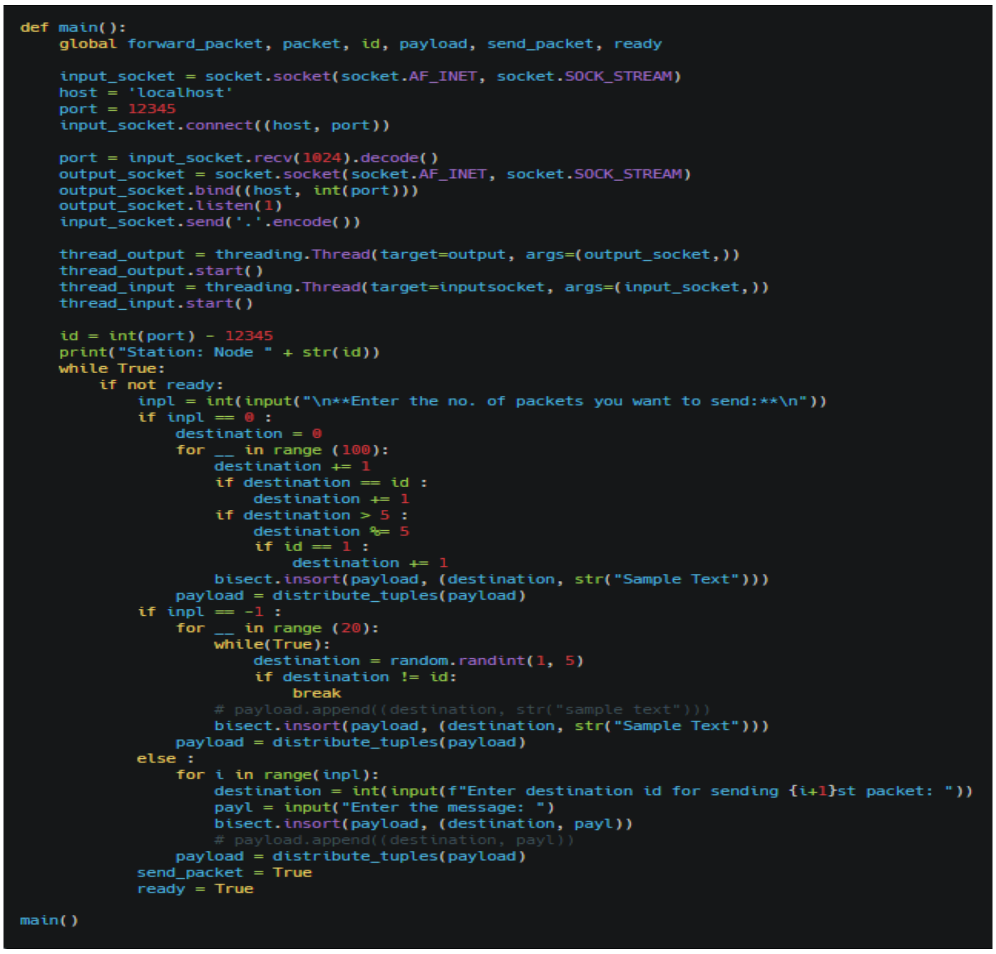
This ‘if’ block handles the case when token arrives back at the sender node and checks if acknowledgement has been received. If not, it retransmits the packets which have not be acked. If all packets have received acknowledgement, then it removes those packets from the buffer. In case, the node holds the token for more than the threshold time, token timeout is signalled and the token is released.



```
        if token["is_taken"] and len(payload):
            token["destination_id"] = int(payload[0][0])
            num_of_packets_to_send = 1
            while len(payload) > num_of_packets_to_send :
                if tht > limit - 0.1 * num_of_packets_to_send:
                    break
                if payload[num_of_packets_to_send][0] == payload[0][0] :
                    num_of_packets_to_send += 1
                else :
                    break
                if num_of_packets_to_send == MAX_NOP:
                    break
            token["num_of_packets"] = num_of_packets_to_send
            token["bitmap"] = "".join(["0" for x in range(num_of_packets_to_send)])
            token["ack"] = False
            waiting_for_token = True
            token["time_sent"] = time.time()
            message = json.dumps(token).encode()

    packet = message
    forward_packet = True
    input_socket.send('.'.encode())
```

Figure 12: input_socket() function part-5

This 'if' block takes care of setting the token parameters like the destination_id, number of packets to be sent, bitmap etc.



```
def main():
    global forward_packet, packet, id, payload, send_packet, ready

    input_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    host = 'localhost'
    port = 12345
    input_socket.connect((host, port))

    port = input_socket.recv(1024).decode()
    output_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    output_socket.bind((host, int(port)))
    output_socket.listen(1)
    input_socket.send('.'.encode())

    thread_output = threading.Thread(target=output, args=(output_socket,))
    thread_output.start()
    thread_input = threading.Thread(target=inputsocket, args=(input_socket,))
    thread_input.start()

    id = int(port) - 12345
    print("Station: Node " + str(id))
    while True:
        if not ready:
            inpl = int(input("\n**Enter the no. of packets you want to send:**\n"))
            if inpl == 0 :
                destination = 0
                for __ in range (100):
                    destination += 1
                    if destination == id :
                        destination += 1
                    if destination > 5 :
                        destination %= 5
                        if id == 1 :
                            destination += 1
                    bisect.insort(payload, (destination, str("Sample Text")))
                payload = distribute_tuples(payload)
            if inpl == -1 :
                for __ in range (20):
                    while(True):
                        destination = random.randint(1, 5)
                        if destination != id:
                            break
                    # payload.append((destination, str("sample text")))
                    bisect.insort(payload, (destination, str("Sample Text")))
                payload = distribute_tuples(payload)
            else :
                for i in range(inpl):
                    destination = int(input(f"Enter destination id for sending {i+1}st packet: "))
                    payl = input("Enter the message: ")
                    bisect.insort(payload, (destination, payl))
                    # payload.append((destination, payl))
                payload = distribute_tuples(payload)
            send_packet = True
            ready = True

main()
```

Figure 13: main() function

4. Results

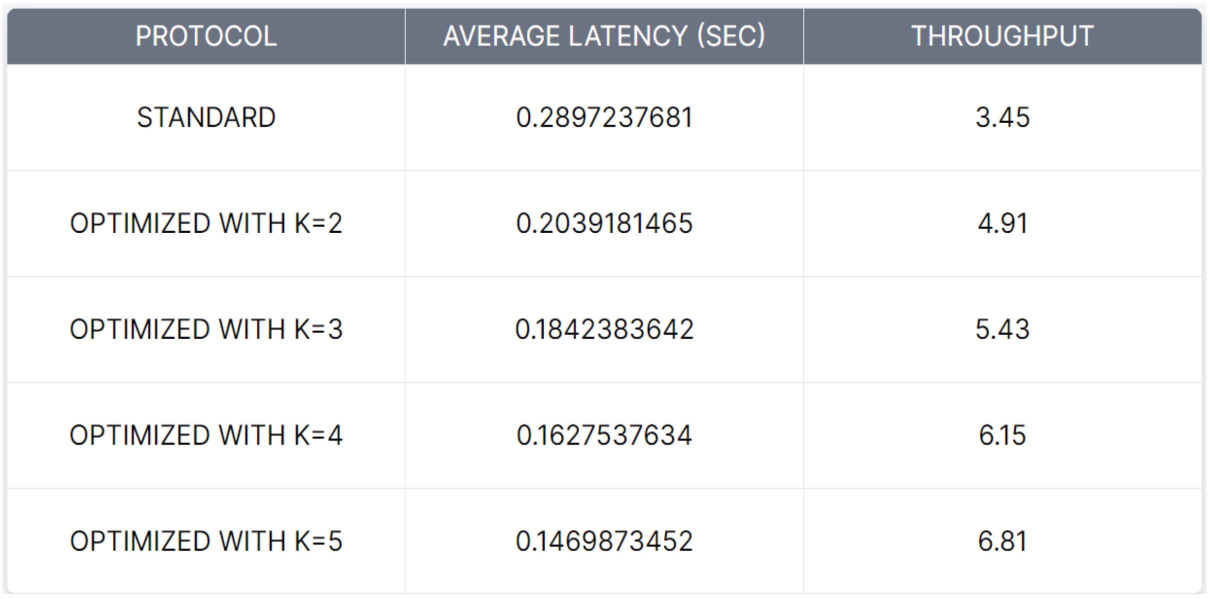
Performance Metrics

The primary metric used to evaluate performance was the total time taken for data transfer per packet, i.e., Latency. We measured this time by recording the timestamp when each packet was sent and when the acknowledgment (ACK) for it was received. This allowed us to calculate the Round-Trip Time (RTT) for the batch of packets, providing insight into the efficiency of the transmission process.

Data Analysis

Data was collected over multiple simulation runs, comparing the performance of the standard protocol and the optimized protocol with 5 nodes in the simulated environment, implemented through terminals/command-line interfaces. The final reported value is the average obtained from all simulation runs. Each node was configured to send 300 packets during the simulation runs. A total of 1500 packets were transmitted during each simulation run.

Comparison



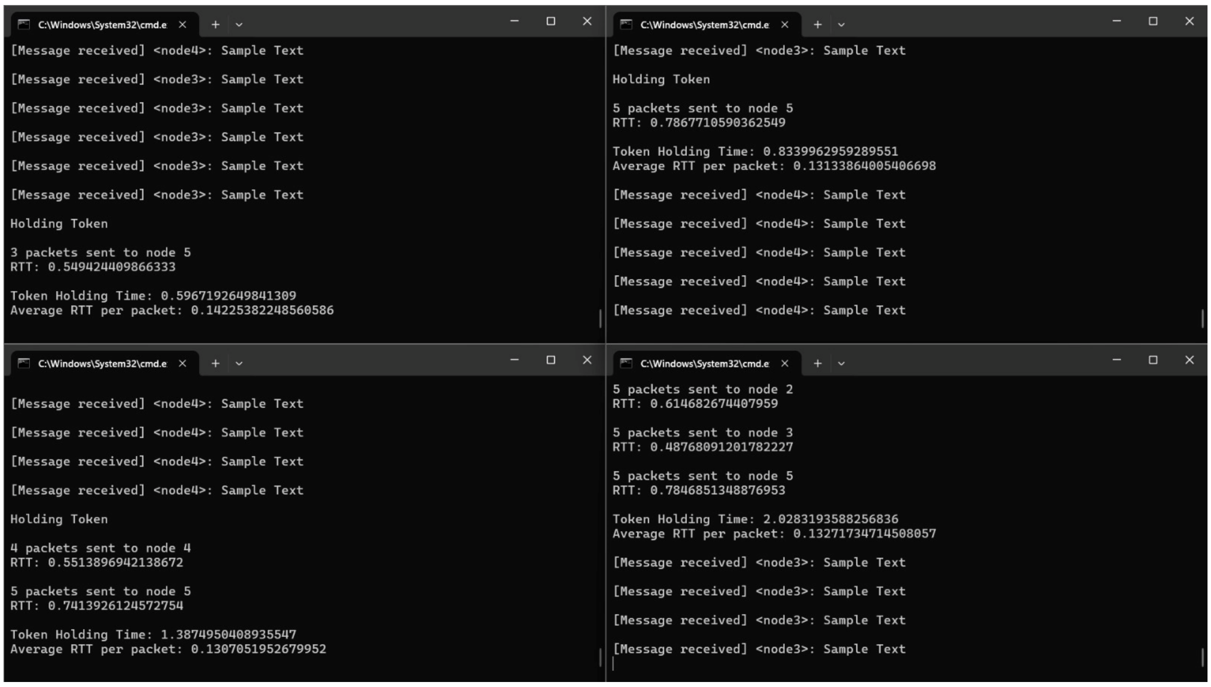
PROTOCOL	AVERAGE LATENCY (SEC)	THROUGHPUT
STANDARD	0.2897237681	3.45
OPTIMIZED WITH K=2	0.2039181465	4.91
OPTIMIZED WITH K=3	0.1842383642	5.43
OPTIMIZED WITH K=4	0.1627537634	6.15
OPTIMIZED WITH K=5	0.1469873452	6.81

Table 1: Average Latency and Throughput Comparison

Here, K represents the maximum number of packets that a sender can transmit in a single batch. Average latency refers to the time taken for data transfer per packet. Throughput is defined as the average number of packets transmitted per second. The standard protocol has K =1 by default.

Screenshot of Results

Optimized Protocol


C:\Windows\System32\cmd.e
C:\Windows\System32\cmd.e
C:\Windows\System32\cmd.e
C:\Windows\System32\cmd.e

```
[Message received] <node4>: Sample Text

[Message received] <node3>: Sample Text

[Message received] <node3>: Sample Text

[Message received] <node3>: Sample Text

[Message received] <node3>: Sample Text

[Message received] <node3>: Sample Text

Holding Token

3 packets sent to node 5
RTT: 0.549424409866333

Token Holding Time: 0.5967192649841309
Average RTT per packet: 0.14225382248560586
```

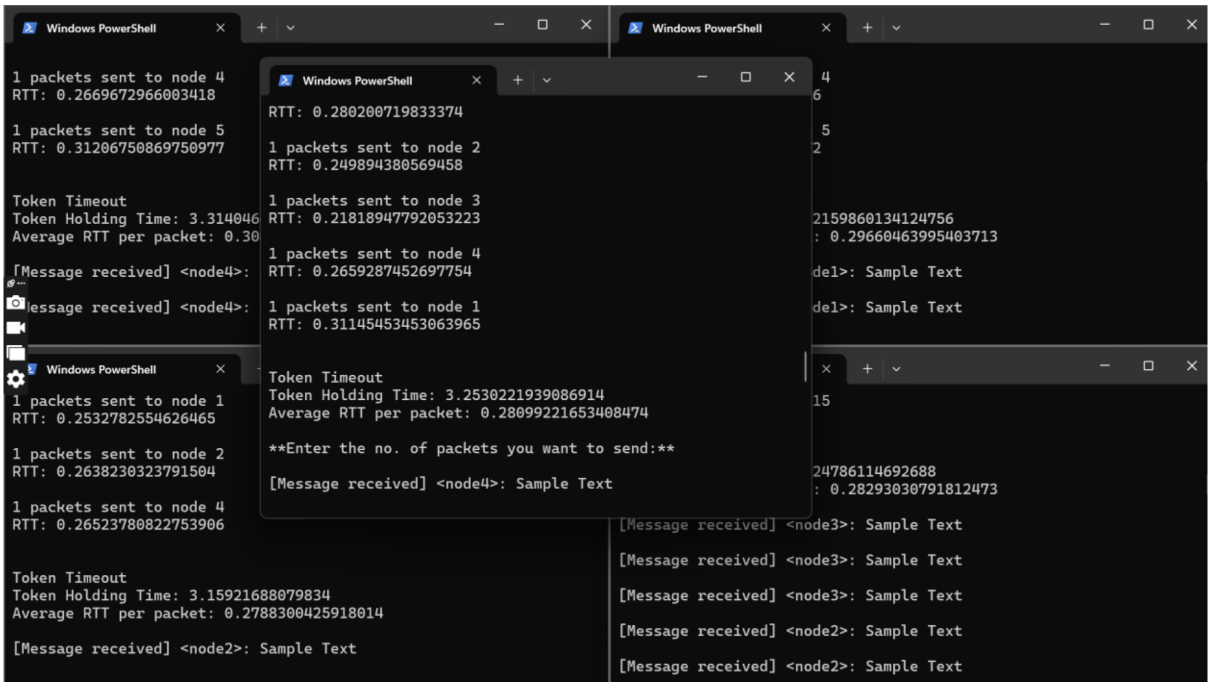
```
[Message received] <node3>: Sample Text

Holding Token

5 packets sent to node 5
RTT: 0.7867710590362549

Token Holding Time: 0.8339962959289551
Average RTT per packet: 0.13133864005406698

[Message received] <node4>: Sample Text

[Message received] <node4>: Sample Text

[Message received] <node4>: Sample Text

[Message received] <node4>: Sample Text

[Message received] <node4>: Sample Text
```

```

[Message received] <node4>: Sample Text

[Message received] <node4>: Sample Text

[Message received] <node4>: Sample Text

[Message received] <node4>: Sample Text

Holding Token

4 packets sent to node 4
RTT: 0.5513896942138672

5 packets sent to node 5
RTT: 0.7413926124572754

Token Holding Time: 1.3874950408935547
Average RTT per packet: 0.1307051952679952
```

```
5 packets sent to node 2
RTT: 0.614682674407959

5 packets sent to node 3
RTT: 0.48768091201782227

5 packets sent to node 5
RTT: 0.7846851348876953

Token Holding Time: 2.0283193588256836
Average RTT per packet: 0.13271734714508057

[Message received] <node3>: Sample Text

[Message received] <node3>: Sample Text

[Message received] <node3>: Sample Text

[Message received] <node3>: Sample Text
```

Figure 14: Results of Optimized Protocol

Standard Protocol



```

1 packets sent to node 4
RTT: 0.2669672966003418

1 packets sent to node 5
RTT: 0.31206750869750977


Token Timeout
Token Holding Time: 3.314046
Average RTT per packet: 0.30

[Message received] <node4>:

lessage received] <node4>:
```

```
RTT: 0.280200719833374

1 packets sent to node 2
RTT: 0.249894380569458

1 packets sent to node 3
RTT: 0.21818947792053223

1 packets sent to node 4
RTT: 0.2659287452697754

1 packets sent to node 1
RTT: 0.3114545345306396 5


Token Timeout
Token Holding Time: 3.2530221939086914
Average RTT per packet: 0.2809922165340847 4

**Enter the no. of packets you want to send:**

[Message received] <node4>: Sample Text
```

```
4
6

5
2


2159860134124756
: 0.29660463995403713

de1>: Sample Text

de1>: Sample Text
```

```
1 packets sent to node 1
RTT: 0.2532782554626465

1 packets sent to node 2
RTT: 0.2638230323791504

1 packets sent to node 4
RTT: 0.26523780822753906


Token Timeout
Token Holding Time: 3.15921688079834
Average RTT per packet: 0.2788300425918014

[Message received] <node2>: Sample Text
```

```
15


24786114692688
: 0.28293030791812473
[Message received] <node3>: Sample Text

[Message received] <node3>: Sample Text

[Message received] <node3>: Sample Text

[Message received] <node2>: Sample Text

[Message received] <node2>: Sample Text
```

Figure 15: Results of Standard Protocol

5. Discussion

Interpretation of Results

The results of the experiments demonstrate a significant reduction in the total time taken for data transfer per packet, with reductions of up to 50% observed in the best-case scenario, following the implementation of the optimization technique. We observed a maximum of 50% increase in throughput and a 50% reduction in average latency. This reduction in time suggests a substantial improvement in efficiency within the network. Specifically, the strategy of streaming packets behind the token emerges as a key factor contributing to this enhanced efficiency. By transmitting multiple packets destined for the same node in a batch, the network experiences less idle time and achieves more streamlined packet delivery. Consequently, this optimization has proven effective in addressing the inefficiencies inherent in the standard protocol, paving the way for smoother and more efficient data transmission.

Limitations

1. Not suitable for two-way communication scenarios, where a sender transmits one packet and then awaits a response from the receiver. In such cases, the optimized protocol's batch transmission approach may introduce delays, as the sender cannot send additional packets until the token is returned. This waiting period can impact real-time or interactive applications where immediate responses are expected, potentially leading to performance degradation or increased latency.

2. In instances where there is only one packet destined for a node, the optimized protocol operates similarly to the standard protocol. Since there are no additional packets to aggregate into a batch, the benefits of batch transmission are not realized. Consequently, the optimization's advantages are limited in scenarios with minimal packet traffic, and the performance remains comparable to that of the standard protocol.

3. Another limitation is that while holding the token, the sender cannot dynamically add more packets into its buffer for transmission. This static nature of packet transmission during token possession may lead to underutilization of network resources, especially in situations where additional packets become available for transmission after the token has been acquired. As a result, the protocol may not fully capitalize on opportunities to optimize data transfer efficiency during periods of token possession.

4. To achieve O(1) time complexity while sending each batch of packets, we arrange the packets in order during the initial addition to the buffer. This approach ensures efficient transmission later but

introduces a preprocessing bottleneck. Consequently, the time spent arranging packets in the buffer initially is critical to maintaining overall efficiency during the sending phase.

These limitations underscore the importance of considering the specific requirements and characteristics of the network environment when implementing and assessing optimization strategies.

6. Conclusion

The optimization of the Ring Token Protocol by streaming multiple packets destined for the same node together has proven to enhance network performance significantly. This approach reduces the overhead of frequent token passing and improves overall network throughput and latency. Future work may explore further enhancements, such as adaptive thresholding for packet grouping and dynamic token management strategies.