Individual Assignment: DATA2410 Reliable Transport Protocol (DRTP) On Top of UDP

Candidate number:

115

Introduction

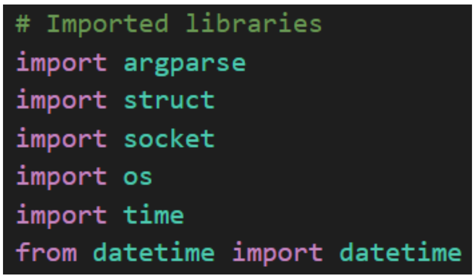
In this project, the goal was to create a reliable file transport protocol using UDP. The code is written in *python* using the advanced text editor *Visual Studio Code*. The project already includes an image called *oak_tree.jpg* to serve as the file to be transferred. The transferred file is written into the *received_file*. Testing of the code was done within *Visual Studio Code*, while the discussion tasks were done with mininet in *Ubuntu* using *Oracle VM VirutelBox* and with the *simple-topo.py* topology. How to run the code is written in the *README* file.

Implementation

All the code is located within the same file: *application.py*. The code consists of 8 functions, here are their order and brief description:

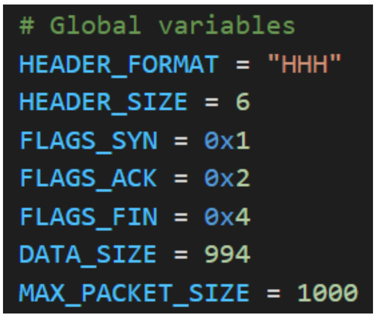
1. **def main()**
   - This function parses custom arguments from the command-line, then calls other functions based on evocation and handles exceptions.
2. **def createPacket()**
   - This function creates a packet with the header and data.
3. **def parsePacket()**
   - This function parses a packet to extract its header and data.
4. **def createSynPacket()**
   - This function creates a SYN packet with the header and file size.
5. **def parseSynPacket()**
   - This function parses a SYN packet to extract its header and file size and handle exception.
6. **def sendPacket()**
   - This function manages the sliding window, controls packet delivery, handles packet resending and sends a packet of data from file to server.
7. **def sendFile()**
   - This function sets up a file transfer system over UDP. It establishes a connection, handles data transmission using sliding window, then starts the connection teardown. It uses timeouts and retransmissions to handle packet loss.
8. **def receiveFile()**
   - This function sets up a UDP server on an IP address and port, then handles the process of receiving a file transmitted over network. It calculates the throughput of received data and can optionally discard packets with a specified sequence number to simulate packet loss.

The main functionality of the application is located within the functions. Outside the functions there is only the following: located at the very top of the code, is all the imported python libraries used in the application.



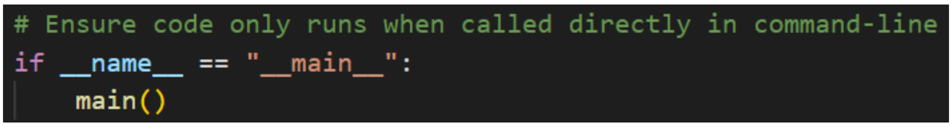
```python
# Imported libraries
import argparse
import struct
import socket
import os
import time
from datetime import datetime
```

Right after the imports, the header format, header size, flags, and data sizes are located. These are based on assignment specifications and are defined as global variables so that they can be accessed anywhere in the code. The "H" in the *HEADER_FORMAT* stands for an unsigned short integer, which is 2 bytes, making the *HEADER_SIZE* 6 bytes long. The 0x1 in the *FLAGS_SYN*, etc., are hexadecimal representations of binary values, meaning that 0x1 = 0001 in binary. The data portion of the packet, *DATA_SIZE* is 994 bytes, while *MAX_PACKET_SIZE* is 1000 bytes, so *DATA_SIZE* + *HEADER_SIZE* = *MAX_PACKET_SIZE*.



```
# Global variables
HEADER_FORMAT = "HHH"
HEADER_SIZE = 6
FLAGS_SYN = 0x1
FLAGS_ACK = 0x2
FLAGS_FIN = 0x4
DATA_SIZE = 994
MAX_PACKET_SIZE = 1000
```

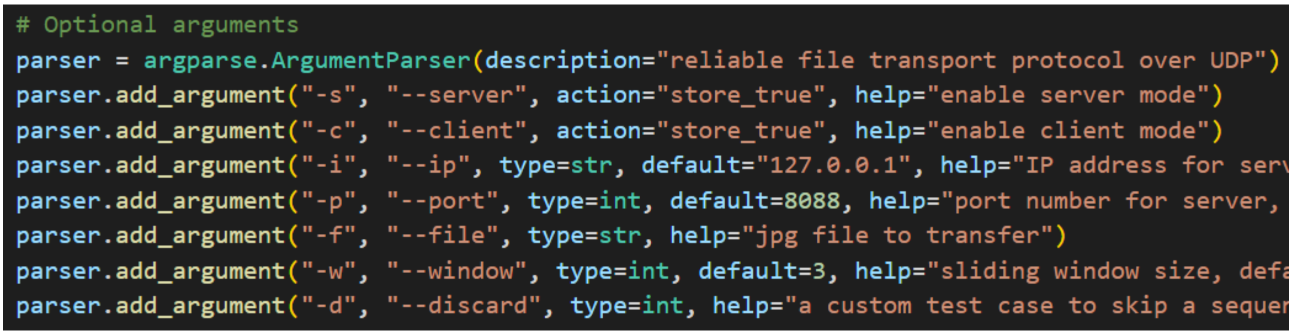
At the very bottom of the code, is a common method for starting the script directly from the command-line, by calling the main() function.



```
# Ensure code only runs when called directly in command-line
if __name__ == "__main__":
    main()
```

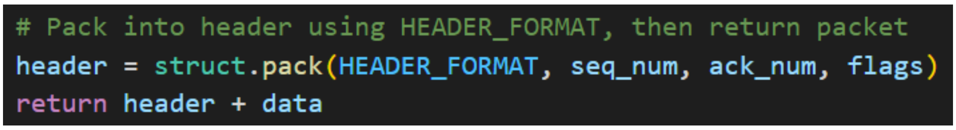
The following is the main functionality of the application.

The argument functionality lies in the main() function. It uses the *argparse* library to define and parse arguments written in the command-line. This way users can insert modes, names, numbers, etc., from the command-line which the code can then use. It also acts as the entry point for the rest of the program with condition logic and handles exceptions.



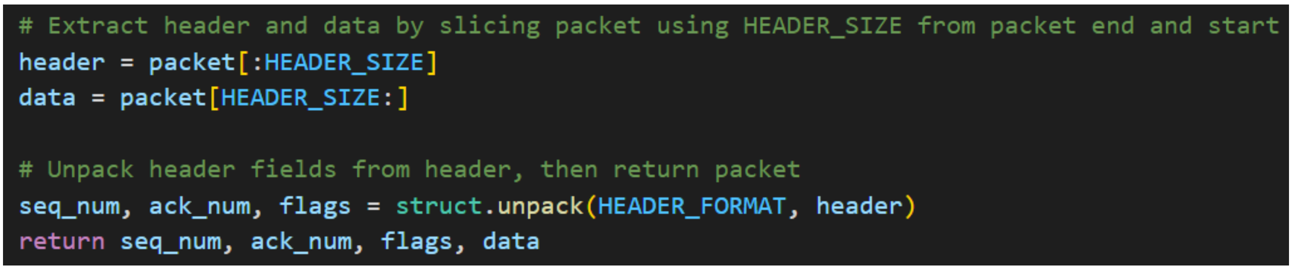
```
# Optional arguments
parser = argparse.ArgumentParser(description="reliable file transport protocol over UDP")
parser.add_argument("-s", "--server", action="store_true", help="enable server mode")
parser.add_argument("-c", "--client", action="store_true", help="enable client mode")
parser.add_argument("-i", "--ip", type=str, default="127.0.0.1", help="IP address for serv
parser.add_argument("-p", "--port", type=int, default=8088, help="port number for server,
parser.add_argument("-f", "--file", type=str, help="jpg file to transfer")
parser.add_argument("-w", "--window", type=int, default=3, help="sliding window size, defa
parser.add_argument("-d", "--discard", type=int, help="a custom test case to skip a sequen
```

All the functions creating packets uses the *struct* library to pack fields and data.



```
# Pack into header using HEADER_FORMAT, then return packet
header = struct.pack(HEADER_FORMAT, seq_num, ack_num, flags)
return header + data
```

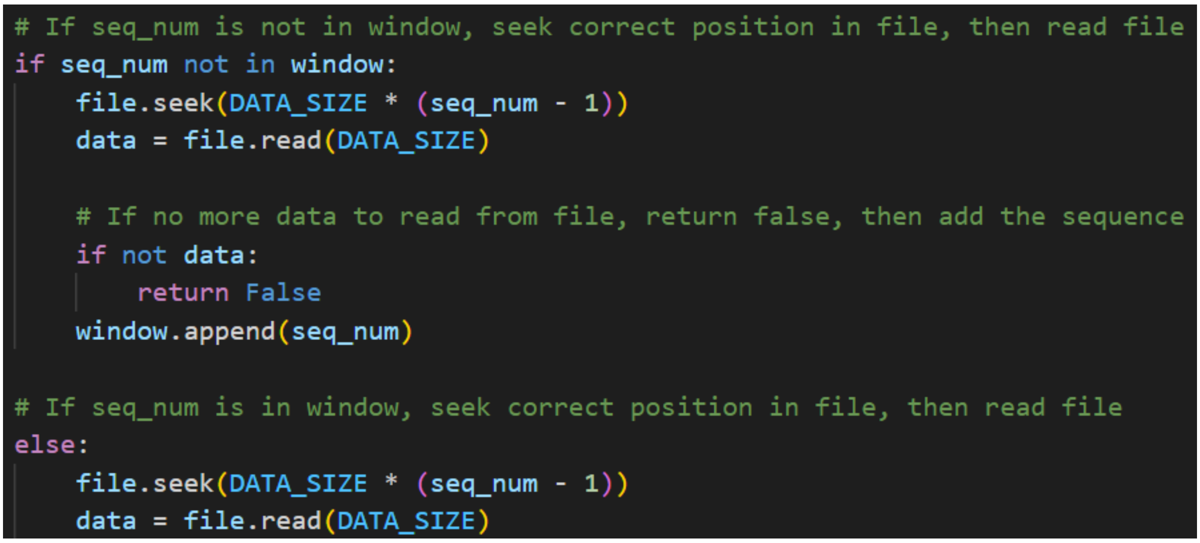
All the functions parsing packets uses slicing to extract header and data from each other and uses the *struct* library to unpack fields and data.



```
# Extract header and data by slicing packet using HEADER_SIZE from packet end and start
header = packet[:HEADER_SIZE]
data = packet[HEADER_SIZE:]

# Unpack header fields from header, then return packet
seq_num, ack_num, flags = struct.unpack(HEADER_FORMAT, header)
return seq_num, ack_num, flags, data
```

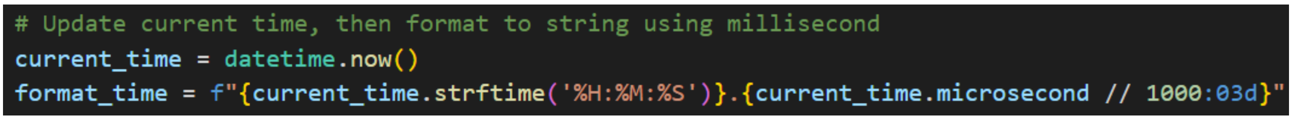
The reason why the synchronize packet has separate functions dedicated for them is because of the inclusion of the *file_size*. I had some issues related to including the *file_size* along with the usual packets because it is not defined in the header format “HHH”, there is no space for it. This is why it is handled before anything else, so that the code can handle normal packets afterwards.

The window sliding and packet delivery functionality lies in the sendPacket() function. It keeps track of sequence numbers for sent packets and add them to *window* if read successfully.



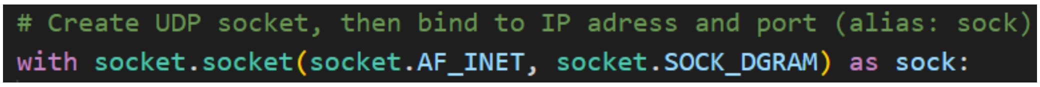
```
# If seq_num is not in window, seek correct position in file, then read file
if seq_num not in window:
    file.seek(DATA_SIZE * (seq_num - 1))
    data = file.read(DATA_SIZE)

    # If no more data to read from file, return false, then add the sequence
    if not data:
        return False
    window.append(seq_num)

# If seq_num is in window, seek correct position in file, then read file
else:
    file.seek(DATA_SIZE * (seq_num - 1))
    data = file.read(DATA_SIZE)
```

This is how the time is formatted and updated with each operation throughout the code.



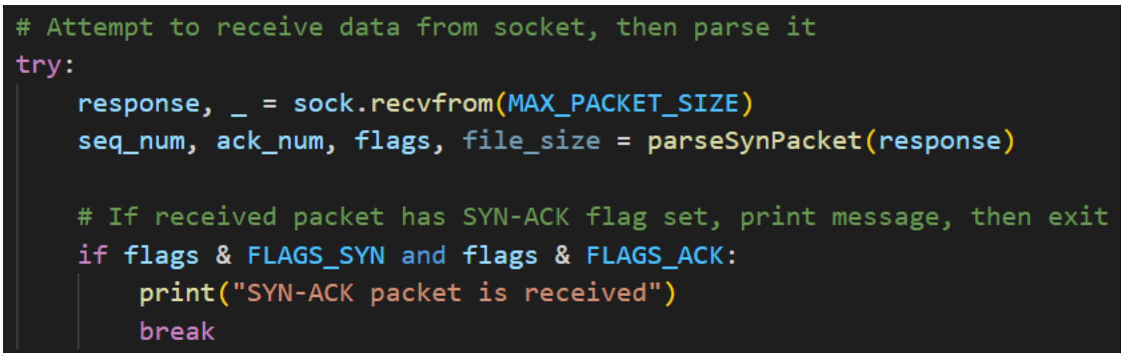
```
# Update current time, then format to string using millisecond
current_time = datetime.now()
format_time = f"{current_time.strftime('%H:%M:%S')}.{current_time.microsecond // 1000:03d}"
```

The client (sender) functionality lies in the sendFile() function. It defines a UDP socket to connect and transmit from.



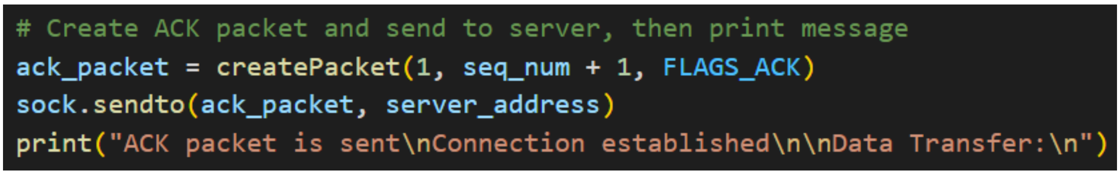
```
# Create UDP socket, then bind to IP adress and port (alias: sock)
with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sock:
```

Then it creates a SYN packet and tries to sends it to the server. Afterwards, it enters a loop (connection establishment phase) waiting to receive a SYN-ACK packet from the server, while also handling exceptions.



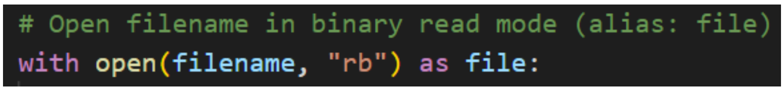
```
# Attempt to receive data from socket, then parse it
try:
    response, _ = sock.recvfrom(MAX_PACKET_SIZE)
    seq_num, ack_num, flags, file_size = parseSynPacket(response)

    # If received packet has SYN-ACK flag set, print message, then exit
    if flags & FLAGS_SYN and flags & FLAGS_ACK:
        print("SYN-ACK packet is received")
        break
```

It then creates and sends an ACK packet to the server to establish the connection.



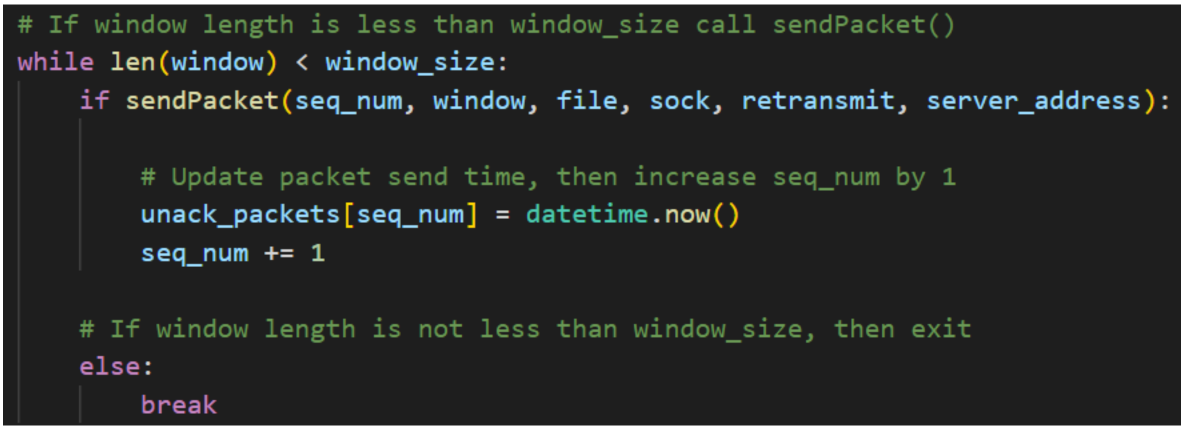
```
# Create ACK packet and send to server, then print message
ack_packet = createPacket(1, seq_num + 1, FLAGS_ACK)
sock.sendto(ack_packet, server_address)
print("ACK packet is sent\nConnection established\n\nData Transfer:\n")
```

It then opens the file in read mode. The file will be utilized later in the code.



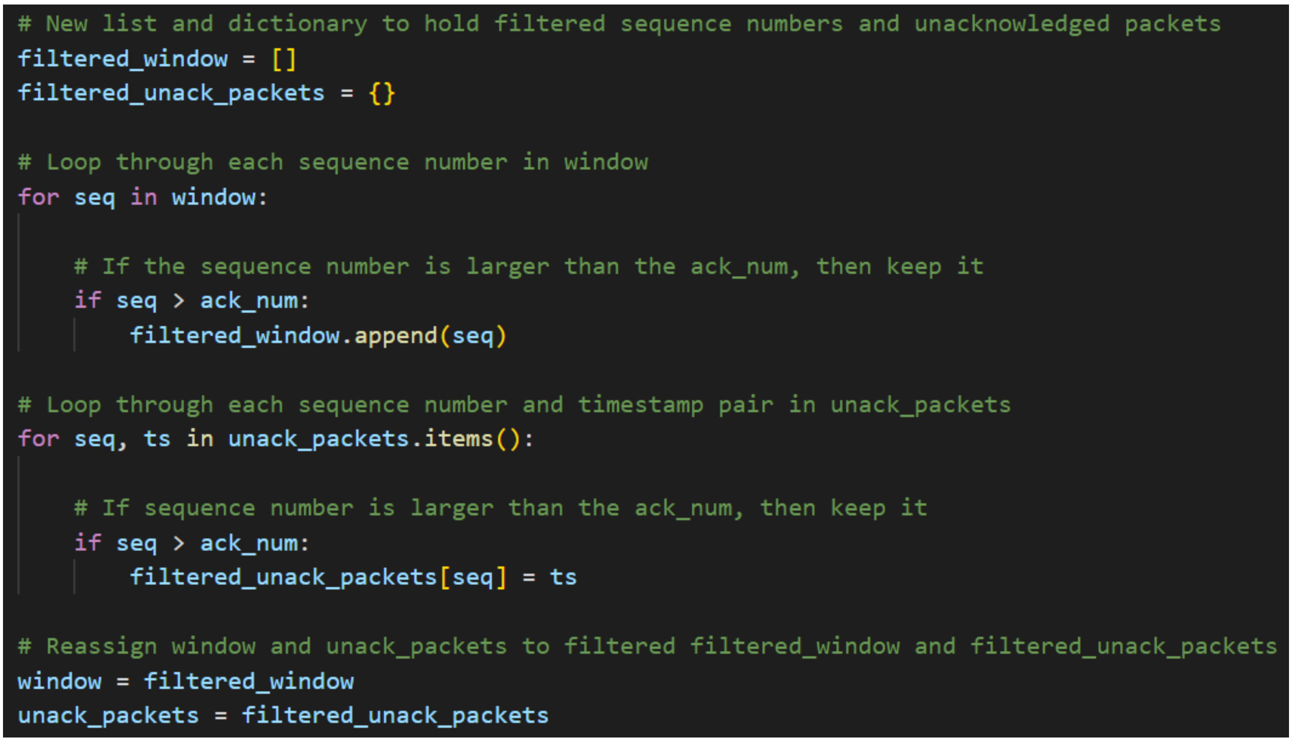
```
# Open filename in binary read mode (alias: file)
with open(filename, "rb") as file:
```

Then it enters a loop to mange the sliding window and retransmissions. When calling the sendPacket() function, it will start sending packets based on the parameters and then exits when it returns false. After each iteration the sequence number is increasing by one.



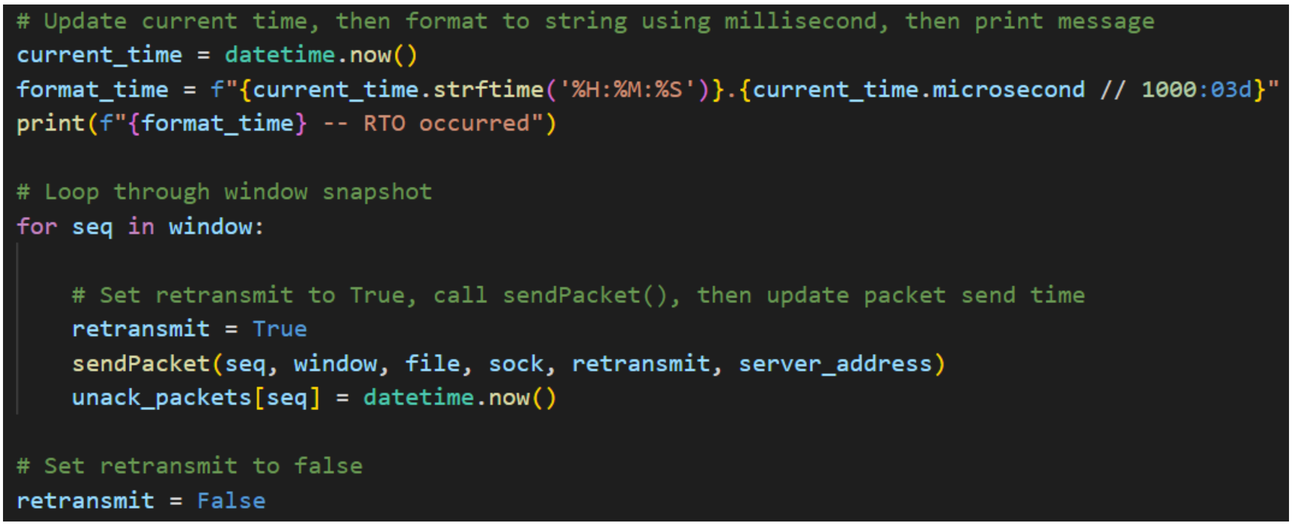
```
# If window length is less than window_size call sendPacket()
while len(window) < window_size:
    if sendPacket(seq_num, window, file, sock, retransmit, server_address):

        # Update packet send time, then increase seq_num by 1
        unack_packets[seq_num] = datetime.now()
        seq_num += 1

    # If window length is not less than window_size, then exit
    else:
        break
```

Part of the previous loop. Filter out sequence numbers and their corresponding unacknowledged packets from *window* and *unack_packets*, then reassign them to *filtered_window* and *filtered_unack_packets*.



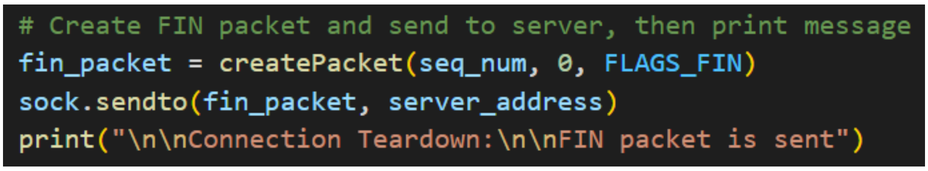
```
# New list and dictionary to hold filtered sequence numbers and unacknowledged packets
filtered_window = []
filtered_unack_packets = {}

# Loop through each sequence number in window
for seq in window:

    # If the sequence number is larger than the ack_num, then keep it
    if seq > ack_num:
        filtered_window.append(seq)

# Loop through each sequence number and timestamp pair in unack_packets
for seq, ts in unack_packets.items():

    # If sequence number is larger than the ack_num, then keep it
    if seq > ack_num:
        filtered_unack_packets[seq] = ts

# Reassign window and unack_packets to filtered filtered_window and filtered_unack_packets
window = filtered_window
unack_packets = filtered_unack_packets
```

Part of the previous loop. A timeout exception to retransmit unacknowledged packets by looping through the window snapshot and sending new packets.



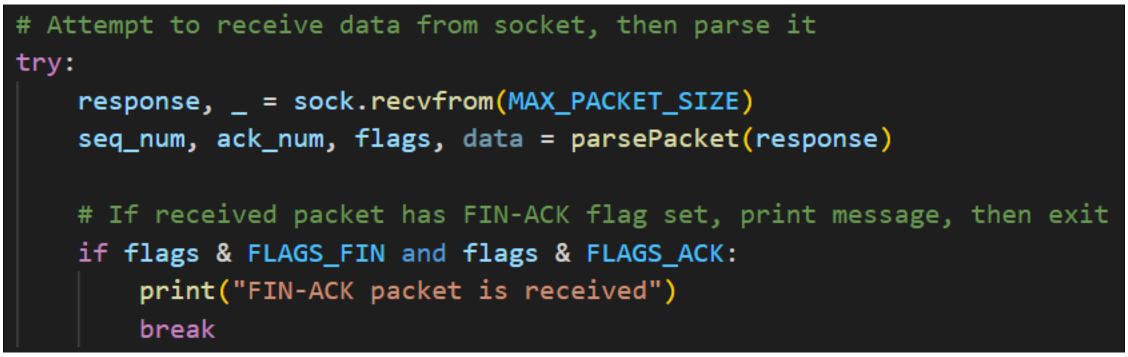
```python
# Update current time, then format to string using millisecond, then print message
current_time = datetime.now()
format_time = f"{current_time.strftime('%H:%M:%S')}.{current_time.microsecond // 1000:03d}"
print(f"{format_time} -- RTO occurred")

# Loop through window snapshot
for seq in window:

    # Set retransmit to True, call sendPacket(), then update packet send time
    retransmit = True
    sendPacket(seq, window, file, sock, retransmit, server_address)
    unack_packets[seq] = datetime.now()

# Set retransmit to false
retransmit = False
```

It then creates and sends a FIN packet to the server to tear down the connection.



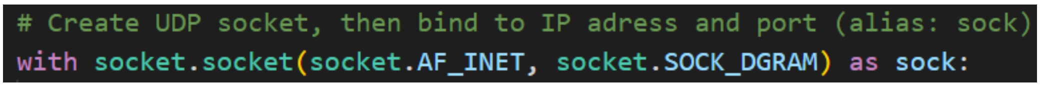
```python
# Create FIN packet and send to server, then print message
fin_packet = createPacket(seq_num, 0, FLAGS_FIN)
sock.sendto(fin_packet, server_address)
print("\n\nConnection Teardown:\n\nFIN packet is sent")
```

Lastly, it enters a loop (connection teardown phase) and tries to wait for a FIN-ACK packet from the server, while also handling exceptions.



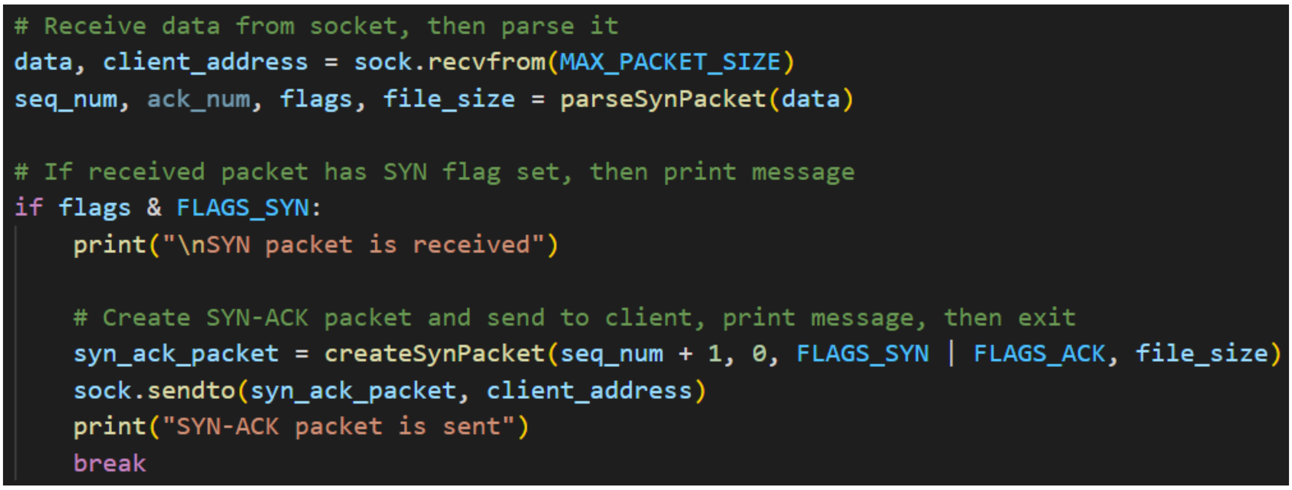
```python
# Attempt to receive data from socket, then parse it
try:
    response, _ = sock.recvfrom(MAX_PACKET_SIZE)
    seq_num, ack_num, flags, data = parsePacket(response)

    # If received packet has FIN-ACK flag set, print message, then exit
    if flags & FLAGS_FIN and flags & FLAGS_ACK:
        print("FIN-ACK packet is received")
        break
```

The server (receiver) functionality lies in the receiveFile() function. It also defines a UDP socket to connect and receive from.



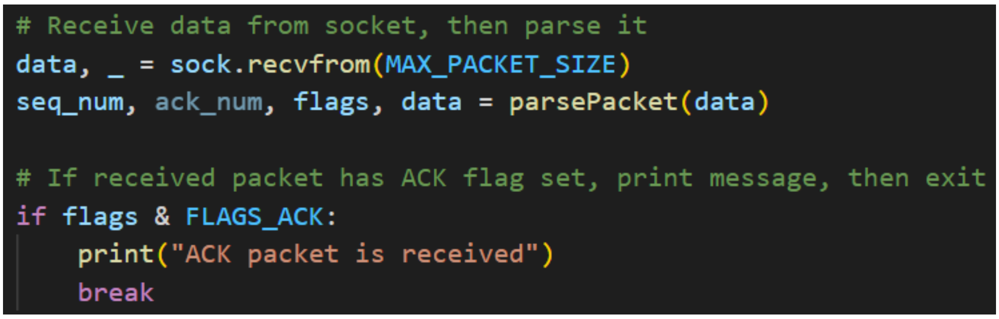
```python
# Create UDP socket, then bind to IP adress and port (alias: sock)
with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sock:
```

It then enters a loop waiting to receive a SYN packet from the client. Afterwards, it creates a SYN-ACK packet and sends it to client.



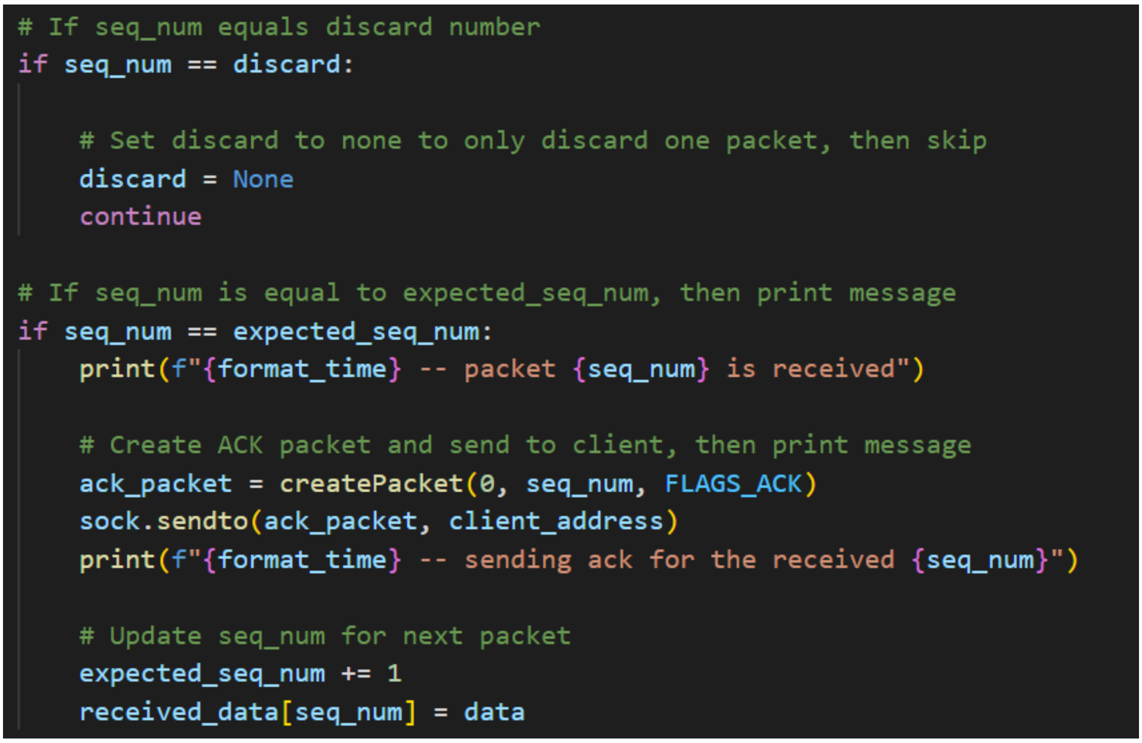
```python
# Receive data from socket, then parse it
data, client_address = sock.recvfrom(MAX_PACKET_SIZE)
seq_num, ack_num, flags, file_size = parseSynPacket(data)

# If received packet has SYN flag set, then print message
if flags & FLAGS_SYN:
    print("\nSYN packet is received")

    # Create SYN-ACK packet and send to client, print message, then exit
    syn_ack_packet = createSynPacket(seq_num + 1, 0, FLAGS_SYN | FLAGS_ACK, file_size)
    sock.sendto(syn_ack_packet, client_address)
    print("SYN-ACK packet is sent")
    break
```

It then waits for an ACK packet from the client to establish the connection.



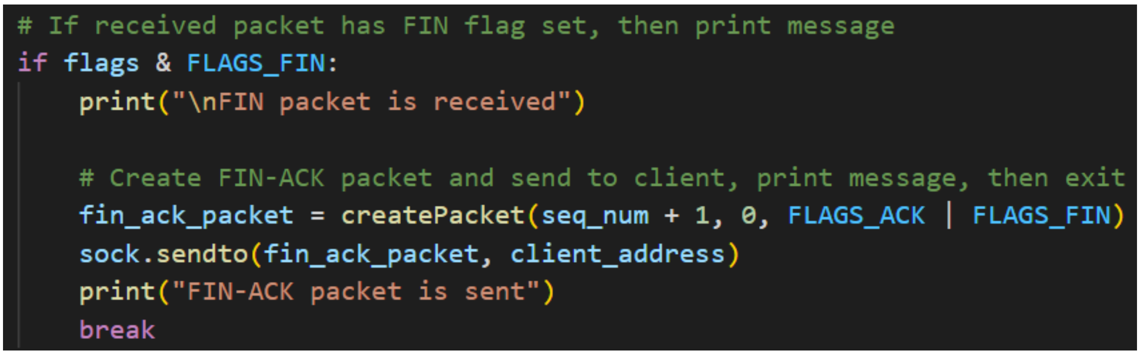
```python
# Receive data from socket, then parse it
data, _ = sock.recvfrom(MAX_PACKET_SIZE)
seq_num, ack_num, flags, data = parsePacket(data)

# If received packet has ACK flag set, print message, then exit
if flags & FLAGS_ACK:
    print("ACK packet is received")
    break
```

Afterwards, it enters a loop that tries to receive data from client and responding with an ACK packet. It also checks if any packets are discarded so it can respond accordingly, while also handling exceptions.



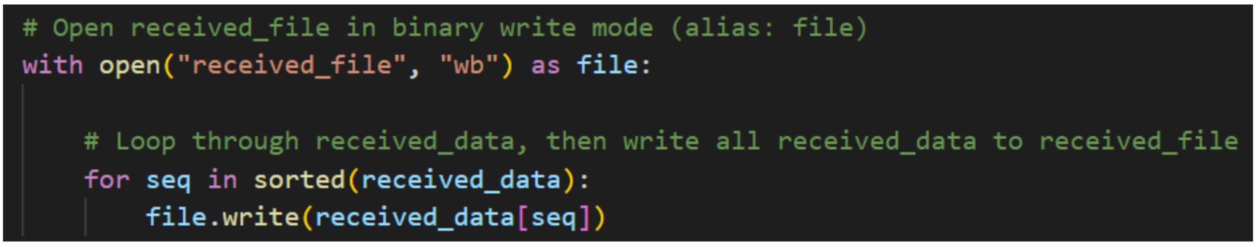
```python
# If seq_num equals discard number
if seq_num == discard:

    # Set discard to none to only discard one packet, then skip
    discard = None
    continue

# If seq_num is equal to expected_seq_num, then print message
if seq_num == expected_seq_num:
    print(f"{format_time} -- packet {seq_num} is received")

    # Create ACK packet and send to client, then print message
    ack_packet = createPacket(0, seq_num, FLAGS_ACK)
    sock.sendto(ack_packet, client_address)
    print(f"{format_time} -- sending ack for the received {seq_num}")

    # Update seq_num for next packet
    expected_seq_num += 1
    received_data[seq_num] = data
```

Then it waits for a FIN packet from the client. Afterwards, it creates a FIN-ACK packet and sends it to client.



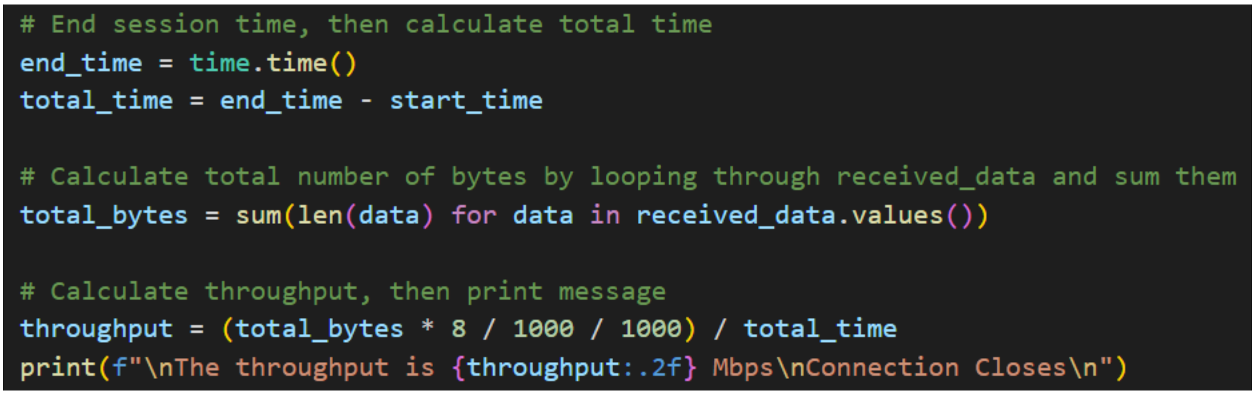
```
# If received packet has FIN flag set, then print message
if flags & FLAGS_FIN:
    print("\nFIN packet is received")

    # Create FIN-ACK packet and send to client, print message, then exit
    fin_ack_packet = createPacket(seq_num + 1, 0, FLAGS_ACK | FLAGS_FIN)
    sock.sendto(fin_ack_packet, client_address)
    print("FIN-ACK packet is sent")
    break
```

Then it opens *received_file* in writing mode and writes the received data to the file by looping through *received_data* and writing in each iteration.



```
# Open received_file in binary write mode (alias: file)
with open("received_file", "wb") as file:

    # Loop through received_data, then write all received_data to received_file
    for seq in sorted(received_data):
        file.write(received_data[seq])
```

Lastly it calculates and displays the throughput based on total data size and total time.



```
# End session time, then calculate total time
end_time = time.time()
total_time = end_time - start_time

# Calculate total number of bytes by looping through received_data and sum them
total_bytes = sum(len(data) for data in received_data.values())

# Calculate throughput, then print message
throughput = (total_bytes * 8 / 1000 / 1000) / total_time
print(f"\nThe throughput is {throughput:.2f} Mbps\nConnection Closes\n")
```

Discussion

1. We see an increase in throughput when increasing the window size because of several reasons. The client must wait for acknowledgements, so with a larger window size the client can transmit more packets before needing an acknowledgement, leading to reduced idle time. Increasing the window size also reduces network latency, that is the time it takes from client to server and back (RTT). Latency effects become less pronounced with larger window sizes because then the client can transmit more packets before needing an acknowledgement. A larger window size also allows more packets to be in transmission at any given time, which better utilizes the network bandwidth capacity leading to larger throughput. Every acknowledgment also requires network resources so by using a larger window size it leads to decreasing the frequency of acknowledgments leading to less overhead and larger throughput.

RTT (default value)	Window size	Throughput
100ms	3	0.21 Mbps
100ms	5	0.34 Mbps
100ms	10	0.73 Mbps

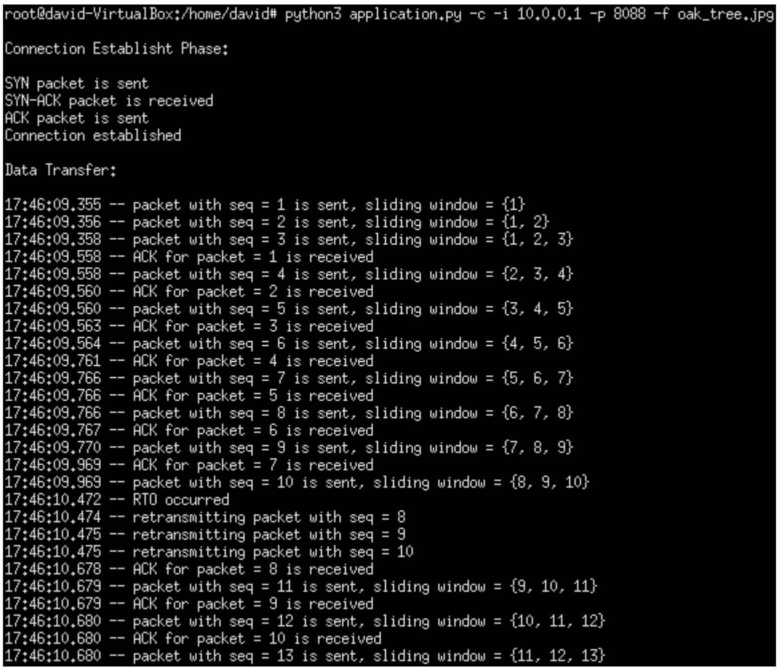
(Throughput is calculated in code)

2. We see an increase in throughput when increasing the window size like before, but we also see the different RTT values effect the throughput as well. With 50ms RTT, the throughput is larger than the first test over the *simple-topo.py* topology. This is because the time it takes from client to server and back is faster, leading to less overhead and better use of bandwidth. Likewise, when the RTT is 200ms, the throughput is the lowest, because it takes more time from client to server and back, leading to more overhead and worse use of bandwidth.

RTT	Window size	Throughput
50ms	3	0.39 Mbps
50ms	5	0.67 Mbps
50ms	10	1.35 Mbps
200ms	3	0.11 Mbps
200ms	5	0.18 Mbps
200ms	10	0.33 Mbps

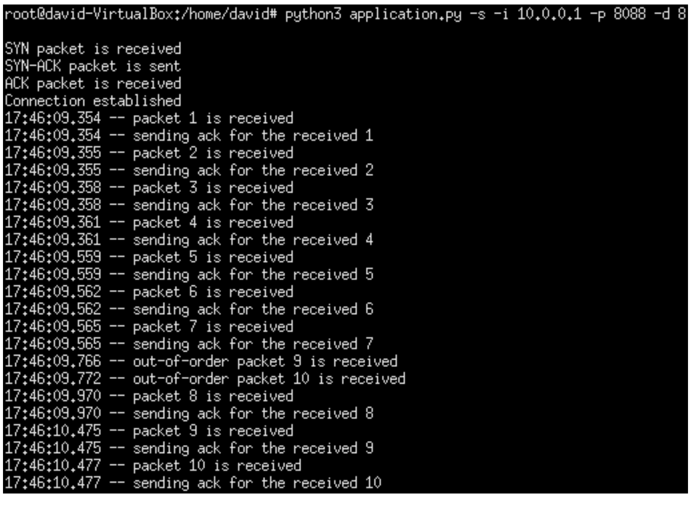
(Throughput is calculated in code)

3. Setting the number 8 with the discard flag; -d 8then it will now skip packet 8, that is the packet with sequence number 8. We see when the code detects the discarded packet has not been acknowledged within the time frame (500ms), it displays a “RTO occurred” message. It then retransmits the discarded packet along with the rest of the packets in the window that have not yet been acknowledged. After confirming that the packets have been received and acknowledged, then the window slides forward so it can receive new packets.



```
root@david-VirtualBox:/home/david# python3 application.py -c -i 10.0.0.1 -p 8088 -f oak_tree.jpg

Connection Establisht Phase:

SYN packet is sent
SYN-ACK packet is received
ACK packet is sent
Connection established

Data Transfer:

17:46:09.355 -- packet with seq = 1 is sent, sliding window = {1}
17:46:09.356 -- packet with seq = 2 is sent, sliding window = {1, 2}
17:46:09.358 -- packet with seq = 3 is sent, sliding window = {1, 2, 3}
17:46:09.558 -- ACK for packet = 1 is received
17:46:09.558 -- packet with seq = 4 is sent, sliding window = {2, 3, 4}
17:46:09.560 -- ACK for packet = 2 is received
17:46:09.560 -- packet with seq = 5 is sent, sliding window = {3, 4, 5}
17:46:09.563 -- ACK for packet = 3 is received
17:46:09.564 -- packet with seq = 6 is sent, sliding window = {4, 5, 6}
17:46:09.761 -- ACK for packet = 4 is received
17:46:09.766 -- packet with seq = 7 is sent, sliding window = {5, 6, 7}
17:46:09.766 -- ACK for packet = 5 is received
17:46:09.766 -- packet with seq = 8 is sent, sliding window = {6, 7, 8}
17:46:09.767 -- ACK for packet = 6 is received
17:46:09.770 -- packet with seq = 9 is sent, sliding window = {7, 8, 9}
17:46:09.969 -- ACK for packet = 7 is received
17:46:09.969 -- packet with seq = 10 is sent, sliding window = {8, 9, 10}
17:46:10.472 -- RTO occurred
17:46:10.474 -- retransmitting packet with seq = 8
17:46:10.475 -- retransmitting packet with seq = 9
17:46:10.475 -- retransmitting packet with seq = 10
17:46:10.678 -- ACK for packet = 8 is received
17:46:10.679 -- packet with seq = 11 is sent, sliding window = {9, 10, 11}
17:46:10.679 -- ACK for packet = 9 is received
17:46:10.680 -- packet with seq = 12 is sent, sliding window = {10, 11, 12}
17:46:10.680 -- ACK for packet = 10 is received
17:46:10.680 -- packet with seq = 13 is sent, sliding window = {11, 12, 13}
```

(Client output)



```
root@david-VirtualBox:/home/david# python3 application.py -s -i 10.0.0.1 -p 8088 -d 8

SYN packet is received
SYN-ACK packet is sent
ACK packet is received
Connection established
17:46:09.354 -- packet 1 is received
17:46:09.354 -- sending ack for the received 1
17:46:09.355 -- packet 2 is received
17:46:09.355 -- sending ack for the received 2
17:46:09.358 -- packet 3 is received
17:46:09.358 -- sending ack for the received 3
17:46:09.361 -- packet 4 is received
17:46:09.361 -- sending ack for the received 4
17:46:09.559 -- packet 5 is received
17:46:09.559 -- sending ack for the received 5
17:46:09.562 -- packet 6 is received
17:46:09.562 -- sending ack for the received 6
17:46:09.565 -- packet 7 is received
17:46:09.565 -- sending ack for the received 7
17:46:09.766 -- out-of-order packet 9 is received
17:46:09.772 -- out-of-order packet 10 is received
17:46:09.970 -- packet 8 is received
17:46:09.970 -- sending ack for the received 8
17:46:10.475 -- packet 9 is received
17:46:10.475 -- sending ack for the received 9
17:46:10.477 -- packet 10 is received
17:46:10.477 -- sending ack for the received 10
```

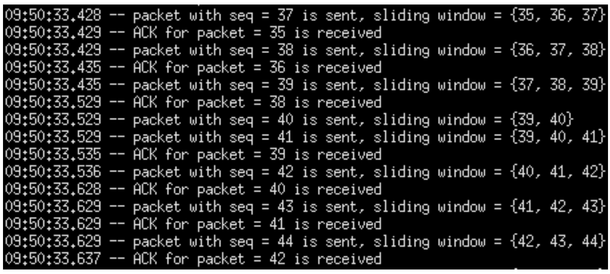
(Server output)

4. Here we run into a limitation with the code. We see the that the packet loss simulation correctly drops the packets within the loss rate. However, the code does not properly react to this event. It does not display the proper RTO occurrence response, nor the retransmitting packet response. This leads to a very similar throughput around 0.20 Mbps. There also seem to be an issue if any packet in the connection establishment phase is dropped, then neither the server nor client will be able to establish the connection. There is most likely some exception handling missing in the connection establishment phase. By the time I discovered these issues, I did not have time to fix them. Nevertheless, here are the results from the packet loss simulation.

We observe that packets within the loss range are dropped, and that the acknowledgment for these packets takes a little longer to be sent and received between the client and server.

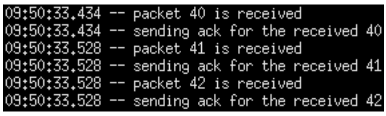
58 packets were received, with a loss rate of 2% gives us around 1 packet lost:

(58 ÷ 100) = (0.58 * 2) = 1.16 ≈ 1



```
09:50:33.428 -- packet with seq = 37 is sent, sliding window = {35, 36, 37}
09:50:33.429 -- ACK for packet = 35 is received
09:50:33.429 -- packet with seq = 38 is sent, sliding window = {36, 37, 38}
09:50:33.435 -- ACK for packet = 36 is received
09:50:33.435 -- packet with seq = 39 is sent, sliding window = {37, 38, 39}
09:50:33.529 -- ACK for packet = 38 is received
09:50:33.529 -- packet with seq = 40 is sent, sliding window = {39, 40}
09:50:33.529 -- packet with seq = 41 is sent, sliding window = {39, 40, 41}
09:50:33.535 -- ACK for packet = 39 is received
09:50:33.536 -- packet with seq = 42 is sent, sliding window = {40, 41, 42}
09:50:33.628 -- ACK for packet = 40 is received
09:50:33.629 -- packet with seq = 43 is sent, sliding window = {41, 42, 43}
09:50:33.629 -- ACK for packet = 41 is received
09:50:33.629 -- packet with seq = 44 is sent, sliding window = {42, 43, 44}
09:50:33.637 -- ACK for packet = 42 is received
```

(Client output)

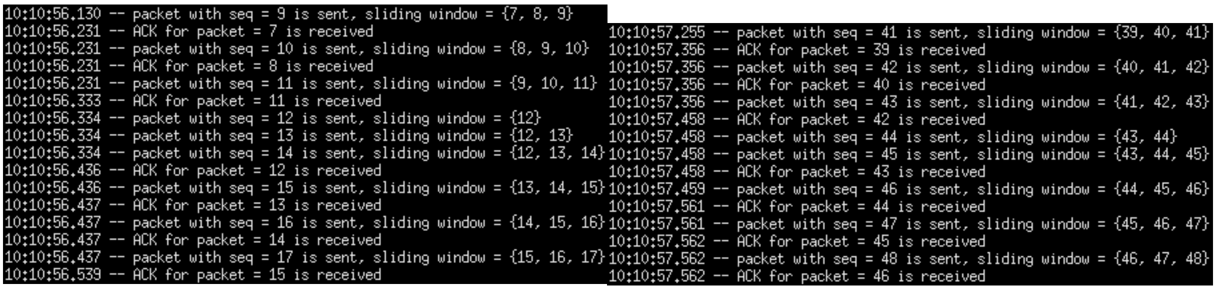


```
09:50:33.434 -- packet 40 is received
09:50:33.434 -- sending ack for the received 40
09:50:33.528 -- packet 41 is received
09:50:33.528 -- sending ack for the received 41
09:50:33.528 -- packet 42 is received
09:50:33.528 -- sending ack for the received 42
```

(Server output)

58 packets were received, with a loss rate of 5% gives us around 3 packets lost:

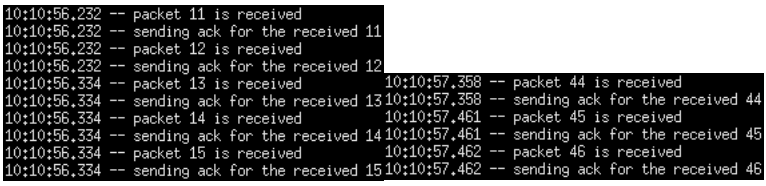
(58 ÷ 100) = (0.58 * 5) = 2.9 ≈ 3



```
10:10:56.130 -- packet with seq = 9 is sent, sliding window = {7, 8, 9}
10:10:56.231 -- ACK for packet = 7 is received
10:10:56.231 -- packet with seq = 10 is sent, sliding window = {8, 9, 10}
10:10:56.231 -- ACK for packet = 8 is received
10:10:56.231 -- packet with seq = 11 is sent, sliding window = {9, 10, 11}
10:10:56.333 -- ACK for packet = 11 is received
10:10:56.334 -- packet with seq = 12 is sent, sliding window = {12}
10:10:56.334 -- packet with seq = 13 is sent, sliding window = {12, 13}
10:10:56.334 -- packet with seq = 14 is sent, sliding window = {12, 13, 14}
10:10:56.436 -- ACK for packet = 12 is received
10:10:56.436 -- packet with seq = 15 is sent, sliding window = {13, 14, 15}
10:10:56.437 -- ACK for packet = 13 is received
10:10:56.437 -- packet with seq = 16 is sent, sliding window = {14, 15, 16}
10:10:56.437 -- ACK for packet = 14 is received
10:10:56.437 -- packet with seq = 17 is sent, sliding window = {15, 16, 17}
10:10:56.539 -- ACK for packet = 15 is received
```

```
10:10:57.255 -- packet with seq = 41 is sent, sliding window = {39, 40, 41}
10:10:57.356 -- ACK for packet = 39 is received
10:10:57.356 -- packet with seq = 42 is sent, sliding window = {40, 41, 42}
10:10:57.356 -- ACK for packet = 40 is received
10:10:57.356 -- packet with seq = 43 is sent, sliding window = {41, 42, 43}
10:10:57.458 -- ACK for packet = 42 is received
10:10:57.458 -- packet with seq = 44 is sent, sliding window = {43, 44}
10:10:57.458 -- packet with seq = 45 is sent, sliding window = {43, 44, 45}
10:10:57.458 -- ACK for packet = 43 is received
10:10:57.459 -- packet with seq = 46 is sent, sliding window = {44, 45, 46}
10:10:57.561 -- ACK for packet = 44 is received
10:10:57.561 -- packet with seq = 47 is sent, sliding window = {45, 46, 47}
10:10:57.562 -- ACK for packet = 45 is received
10:10:57.562 -- packet with seq = 48 is sent, sliding window = {46, 47, 48}
10:10:57.562 -- ACK for packet = 46 is received
```

(Client output)



```
10:10:56.232 -- packet 11 is received
10:10:56.232 -- sending ack for the received 11
10:10:56.232 -- packet 12 is received
10:10:56.232 -- sending ack for the received 12
10:10:56.334 -- packet 13 is received
10:10:56.334 -- sending ack for the received 13
10:10:56.334 -- packet 14 is received
10:10:56.334 -- sending ack for the received 14
10:10:56.334 -- packet 15 is received
10:10:56.334 -- sending ack for the received 15
```

```
10:10:57.358 -- packet 44 is received
10:10:57.358 -- sending ack for the received 44
10:10:57.461 -- packet 45 is received
10:10:57.461 -- sending ack for the received 45
10:10:57.462 -- packet 46 is received
10:10:57.462 -- sending ack for the received 46
```

(Server output)

References

Safiqul, I. (2023, February 6th). *2410*. GitHub. https://github.com/safiqul/2410/tree/main

Solomon, A. (2021, October 20th). Medium. *UDP protocol with a header implementation in python.* https://abdesol.medium.com/udp-protocol-with-a-header-implementation-in-python-b3d8dae9a74b

Baeldung., & Aibin, M. (2024). Baeldung. *Go-Back-N Protocol*. https://www.baeldung.com/cs/networking-go-back-n-protocol