PROJECT README

Kabirsingh Karamjeetsingh Bhatia (kbhatia)

Prabhudatta Mishra (pmishra4)

Readme Structure:
We have extended the previous readme from project 1 with the requirements for project 2 . We have highlighted the headings with blue for the new content that we have added.

Project Initiation:

Our Project has two code files auc_server_rdt.py and auc_client_rdt.py. The process to run the two files is simple.

Step:01 Run pip3 install -r requirements.txt

Step:02 Run python3 auc_server_rdt.py <host> <port> on one terminal to start the auctioneer server

```
       valid_lft forever preferred_lft forever
[[pmishra4@vclvm179-9 Auction_Socket_Programming]$ python3 auc_server_rdt.py  10.40.131.9 3000
Auctioneer is ready for hosting auctions!
```

Step:03 Run python3 auc_client_rdt.py <host> <port> <rdtport> <packet_loss_rate>
<host> = IP address of the server
<port> = port of the server
<rdtport> = the rdt port no. for the clients for file transfer

<packet_loss_rate> (optional) = Specify the range of packet loss rate in range [0,1]. Default value is 0 (no packet loss) if argument is not provided.

We are sending a smaller file for demonstration purposes. We used a larger file of 2MB for performance measurement.

For normal case we have taken 0 for packet loss rate

The first client will join as a Seller

```
auc_client_rdt.py: error: the following arguments are required: packet_loss_rate
[[pmishra4@vclvm179-57 Auction_Socket_Programming]$ python3 auc_client_rdt.py  10.40.131.9 3000 3001 0
Connecting to server at 10.40.131.9:3000...
Server: Your role is: [Seller]
Please submit auction request:

Enter auction type, minimum price, maximum number of bids, and item name (separated by spaces):
```

Testing the setup:

Auction: Type 1
Packet_loss: 0

As discussed earlier we have started a server and a client has already joined as a seller.

Step: 01 Seller submits auction request

```
[[pmishra4@vclvm179-57 Auction_Socket_Programming]$ python3 auc_client_rdt.py  10.40.131.9 3000 3001 0
Connecting to server at 10.40.131.9:3000...
Server: Your role is: [Seller]
Please submit auction request:

[Enter auction type, minimum price, maximum number of bids, and item name (separated by spaces): 1 20 2 wolfpack
Auction request sent to server.
```

Server screen:

```
[[pmishra4@vclvm179-9 Auction_Socket_Programming]$ python3 auc_server_rdt.py  10.40.131.9 3000
 Auctioneer is ready for hosting auctions!
 New Seller is connected from 10.40.131.57:36910
 >> New Seller Thread spawned
 Action request received. Now waiting for Buyer
```

Step:02 Buyers join the auction

Buyer:1

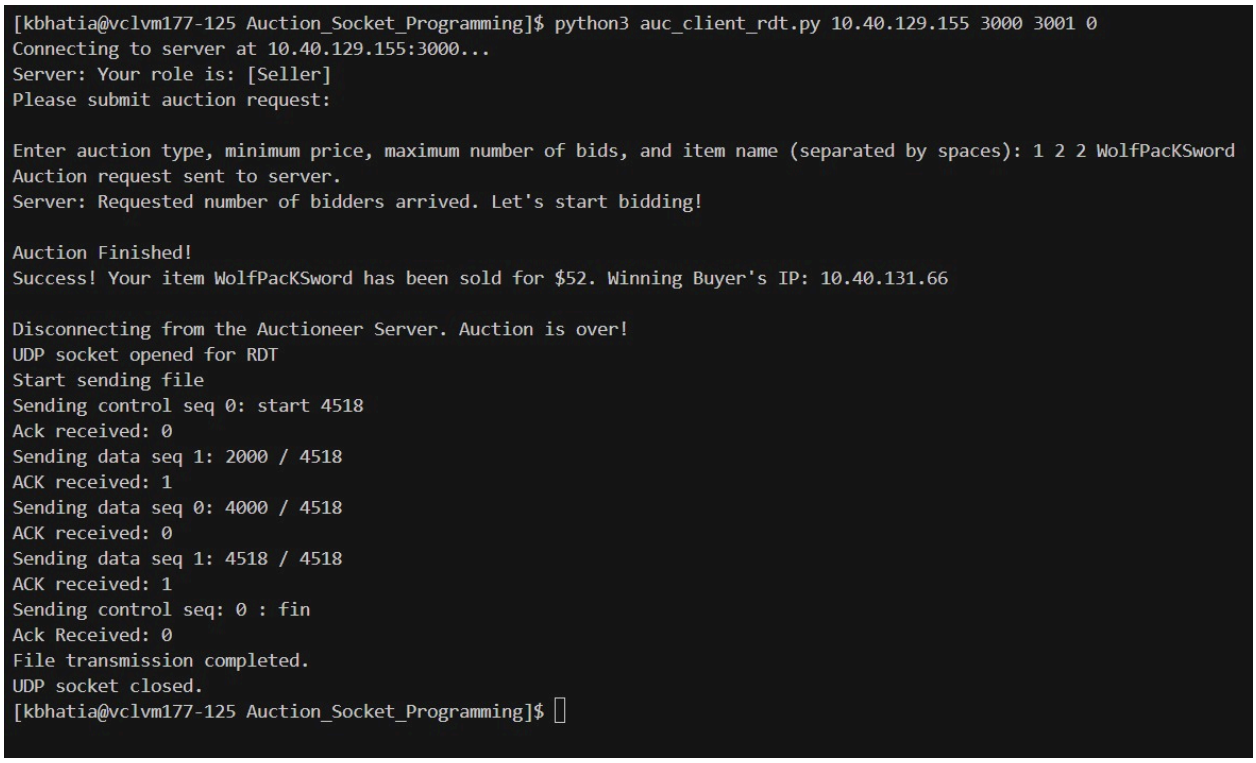
```
 Successfully installed numpy-1.19.5
[[pmishra4@vclvm177-155 Auction_Socket_Programming]$ python3 auc_client_rdt.py  10.40.131.9 3000 3003 0
 Connecting to server at 10.40.131.9:3000...
 Server: Your role is: [Buyer]

 Server: Requested number of bidders arrived. Let's start bidding!

 Server: Please submit your bid:
 Enter bid:
```

Buyer:2

```
[pmishra4@vclvm179-68 Auction_Socket_Programming]$ python3 auc_client_rdt.py  10.40.131.9 3000 3002 0
Connecting to server at 10.40.131.9:3000...
Server: Your role is: [Buyer]

Server: The Auctioneer is still waiting for other Buyer to connect...

Server: Requested number of bidders arrived. Let's start bidding!

Server: Please submit your bid:
Enter bid:
```

Both the buyers make their bids and the auction concludes

Seller’s Screen:



As you can check that the Seller receives the Buyer’s IP from the server and starts sending the file through the rdt port.
An UDP socket gets opened and the seller starts the communication by sending a start message with seq no. 0 and then it receives an ACK for the start message from the winning Buyer and then it starts sending the file to the winning buyer. After completely sending the file, the seller sends a fin message with seq no. 0 and the winning buyer responds with an ACK and the UDP socket is closed.

Without packet loss, the Seller sends out the file correctly and gets the feedback from WB correctly

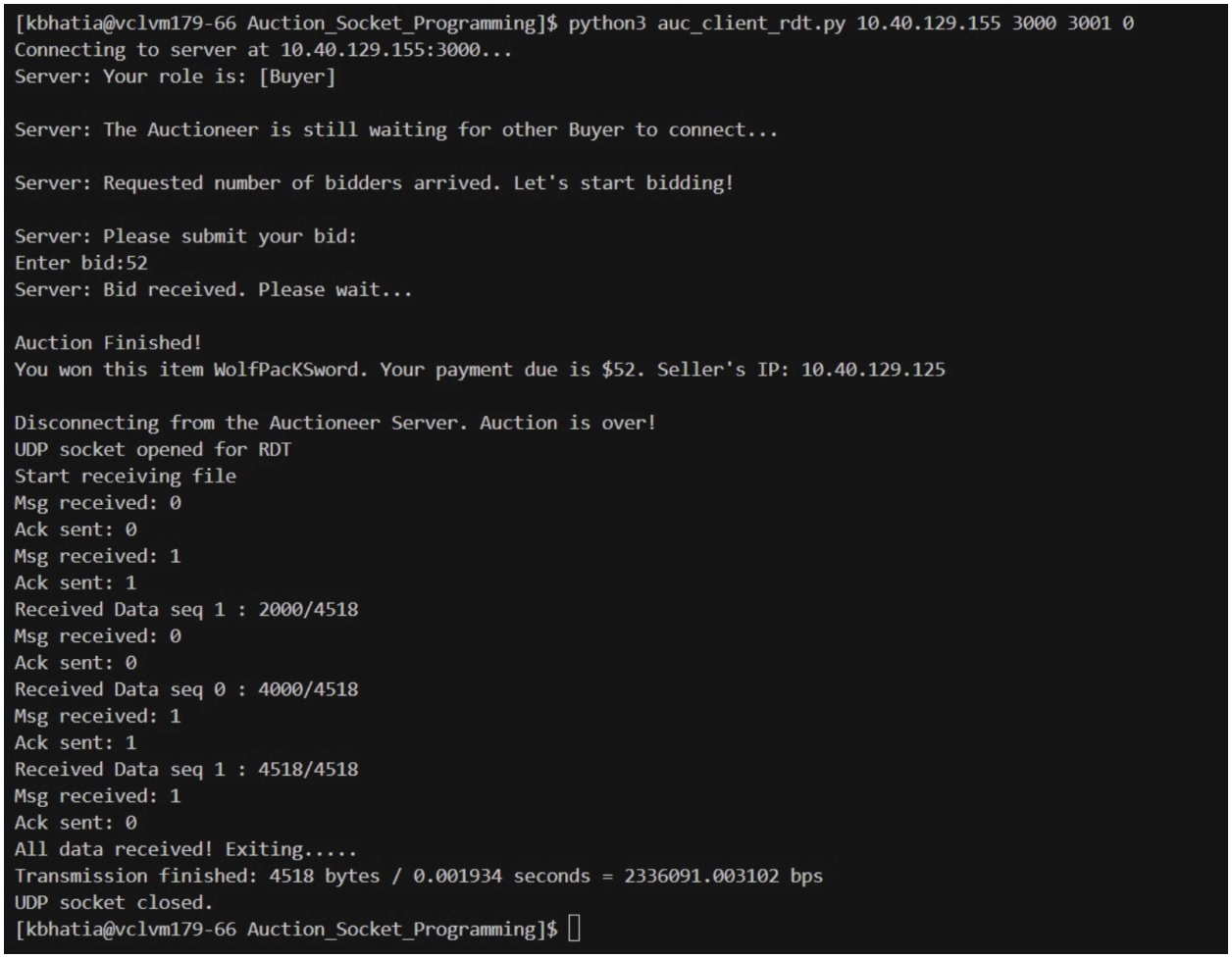
Server distributes the IP address of Seller and WB correctly

CODE PROOF:

```
    - price: Winning bid amount

    """
    winner_conn = next(conn for conn, buyer_id in self.buyers if buyer_id == winner_id)
    seller_ip = self.seller_conn.getpeername()[0]
    buyer_ip = winner_conn.getpeername()[0]

    winner_conn.sendall(f"Auction Finished!\nYou won this item {self.auction_details['item_name']}. Your payment du
    self.seller_conn.sendall(f"Auction Finished!\nSuccess! Your item {self.auction_details['item_name']} has been s

    print(f"The item was sold to {winner_id} for ${price}")

    for conn, buyer_id in self.buyers:  # Notify losing bidders about their unsuccessful attempts
        if buyer_id != winner_id:
            conn.sendall(b"Server: Unfortunately, you did not win in the last round.\n")
```

Since this is the normal case scenario there is no packet loss during the file transfer process.

Winner Buyer Screen:



After the auction gets over the winning buyer receives a message stating that they have won along with the Seller's IP address from the server. Then a UDP socket is opened and the winning buyer starts receiving the file. It sends an ACK with the same seq number as that of the Msg received. After the entire file is obtained, the buyer receives a fin message and stops the process

by displaying 1) number of bytes received, 2) the time taken to receive all bytes (TCT), and 3) the average throughput (AT).

Without packet loss, the WB receives the file correctly and gets the feedback from Seller correctly: get the message from Seller, send out Ack.

Loser Buyer Screen:

```
[kbhatia@vclvm178-238 Auction_Socket_Programming]$ python3 auc_client_rdt.py 10.40.129.155 3000 3001 0
Connecting to server at 10.40.129.155:3000...
Server: Your role is: [Buyer]

Server: Requested number of bidders arrived. Let's start bidding!

Server: Please submit your bid:
Enter bid:25
Server: Bid received. Please wait...

Server: Unfortunately, you did not win in the last round.
```

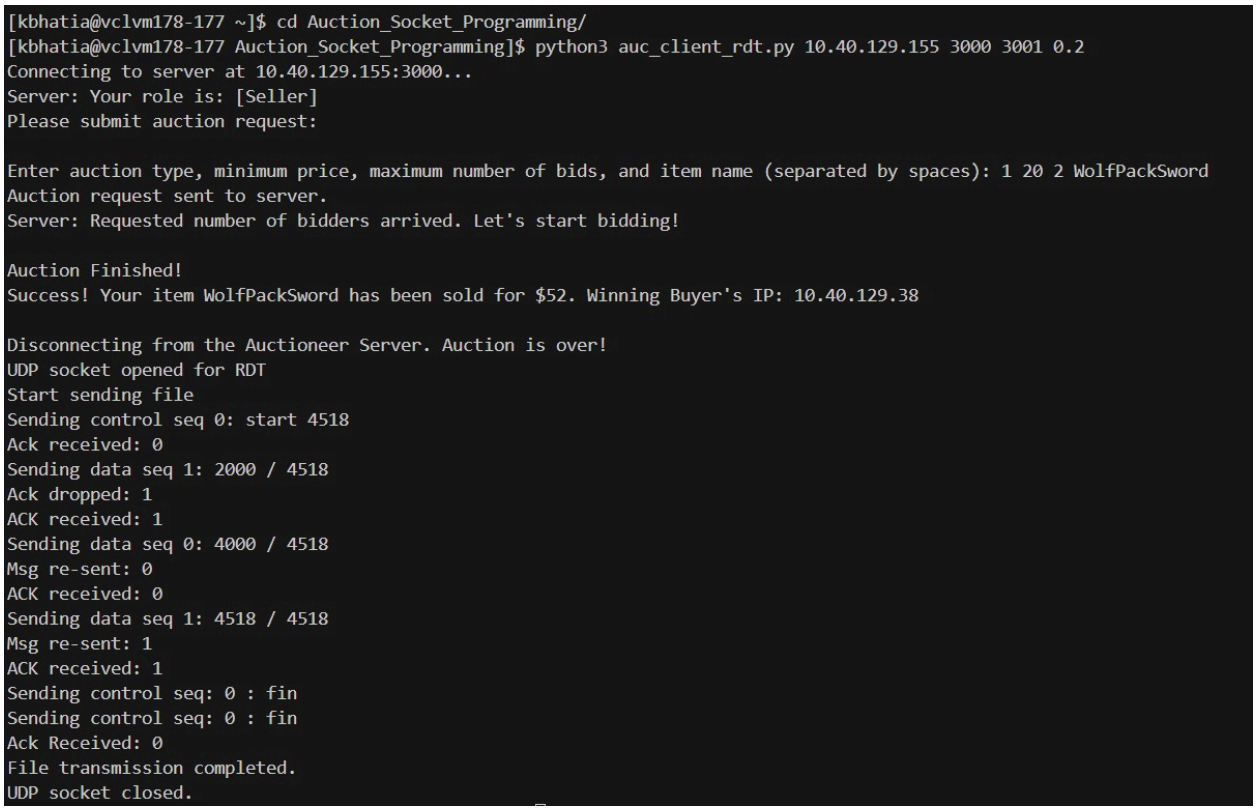
The loser buyer gets a message stating that they have lost the auction from the server.

Auction: Type 1
Packet_loss: 0.2

Now moving on from the normal case, we set the packet loss rate at 0.2 which gives a probability of a packet loss being 20%.

We followed the exact same process as above, but now we observed that there was packet loss while the file transferring between the seller and the buyer.

Seller’s Screen (during packet loss)



As we are simulating the packet loss of the Acknowledgement from the winning buyer, you can see that the start message was sent and an ack was received, after that the data with seq 1 was sent. Seller starts the file transfer sending process correctly.

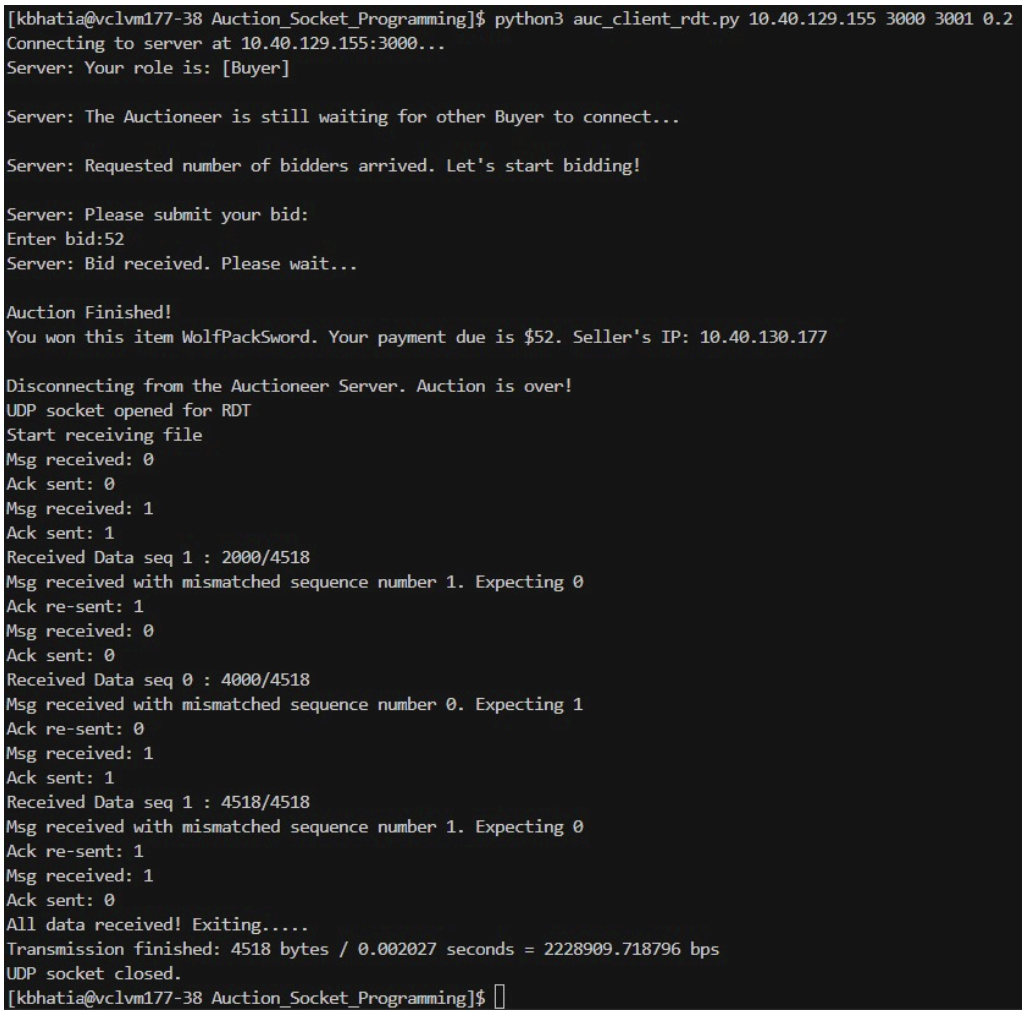
Now we **simulated Packet loss** and you can see that the ACK for 1 was dropped which made the seller wait for receiving the ACK. After the seller receives the correct ACK it changes its state

to 0 and starts sending the file. If a timeout occurs then the Seller resends the packet as observed msg resent : 1.
After transferring all the packets the seller sends a fin message and waits for its acknowledgement, after getting the Ack it closes the UDP socket.

With packet loss, the Seller responds to the packet drop correctly.

Seller terminates the sending process when file transfer completes

Buyer’s Screen (After packet loss)



After the winning buyer receives the IP address of the seller from the server it creates a UDP socket and checks for the packets coming from the seller.

Since we are simulating packet loss, here we have simulated that the packets coming from the sender are getting dropped. If the Buyer receives any out of order packets from the sender it discards them and waits for the correct packet with the correct seq no. to arrive and resends the ack with the expected seq no.

After receiving the correct packet from the sender with the expected seq no. , the buyer changes it's states and continues the process. After receiving the entire file. The integrity of the file is checked with the checksum it received in the start message, compares the checksum and verifies the received.file.

FILE INTEGRITY:

```
# print("File received and saved as 'received.file'")
## creating checksum for the received data
received_checksum = cal_check_sum('received.file')

if received_checksum == original_checksum :
    print("All data received! Exiting.....")
    throughput = get_average_throughput(current_size, transfer_completion_time)
    print(f"Transmission finished: {current_size} bytes / {transfer_completion_time} seconds = {throughput} bps")
else:
    print("File transfer is complete and the file is corrupted")
```

WB receives the whole file at the end of file transfer and the file can be opened successfully

After sending the ack for the fin message, the UDP socket is closed and the buyer displays the1) number of bytes received, 2) the time taken to receive all bytes (TCT), and 3) the average throughput (AT).

Auction: Type 2

Packet_loss: 0.2

We had an error in the first Project :
FEEDBACK: **When testing the Type 2 auction, if the minimum price is set to 100, and one client submits a bid of 99 while another submits a bid of 110, the item should be sold at the minimum price of 100 because 100 itself is also considered a valid bid.**

So we tested out the same scenario but introduced a packet loss for the UDP transfer.

Seller's Screen:

```
[kbhatia@vclvm178-177 Auction_Socket_Programming]$ python3 auc_client_rdt.py 10.40.129.155 3000 3001 0.2
Connecting to server at 10.40.129.155:3000...
Server: Your role is: [Seller]
Please submit auction request:

Enter auction type, minimum price, maximum number of bids, and item name (separated by spaces): 2 100 2 WolfPackSword
Auction request sent to server.
Server: Requested number of bidders arrived. Let's start bidding!

Auction Finished!
Success! Your item WolfPackSword has been sold for $100. Winning Buyer's IP: 10.40.129.38

Disconnecting from the Auctioneer Server. Auction is over!
UDP socket opened for RDT
Start sending file
Sending control seq 0: start 4518
Ack dropped: 0
Ack received: 0
Sending data seq 1: 2000 / 4518
ACK received: 1
Sending data seq 0: 4000 / 4518
ACK received: 0
Sending data seq 1: 4518 / 4518
ACK received: 1
Sending control seq: 0 : fin
Ack Received: 0
File transmission completed.
UDP socket closed.
[kbhatia@vclvm178-177 Auction_Socket_Programming]$
```

Since we set the price as 100 by the seller. We gave two bids 99, 110 and the highest bidder got the item at 100. Now we started the UDP socket at the seller side and simulated a packet loss at 0.2 and the same process took place as we discussed above.

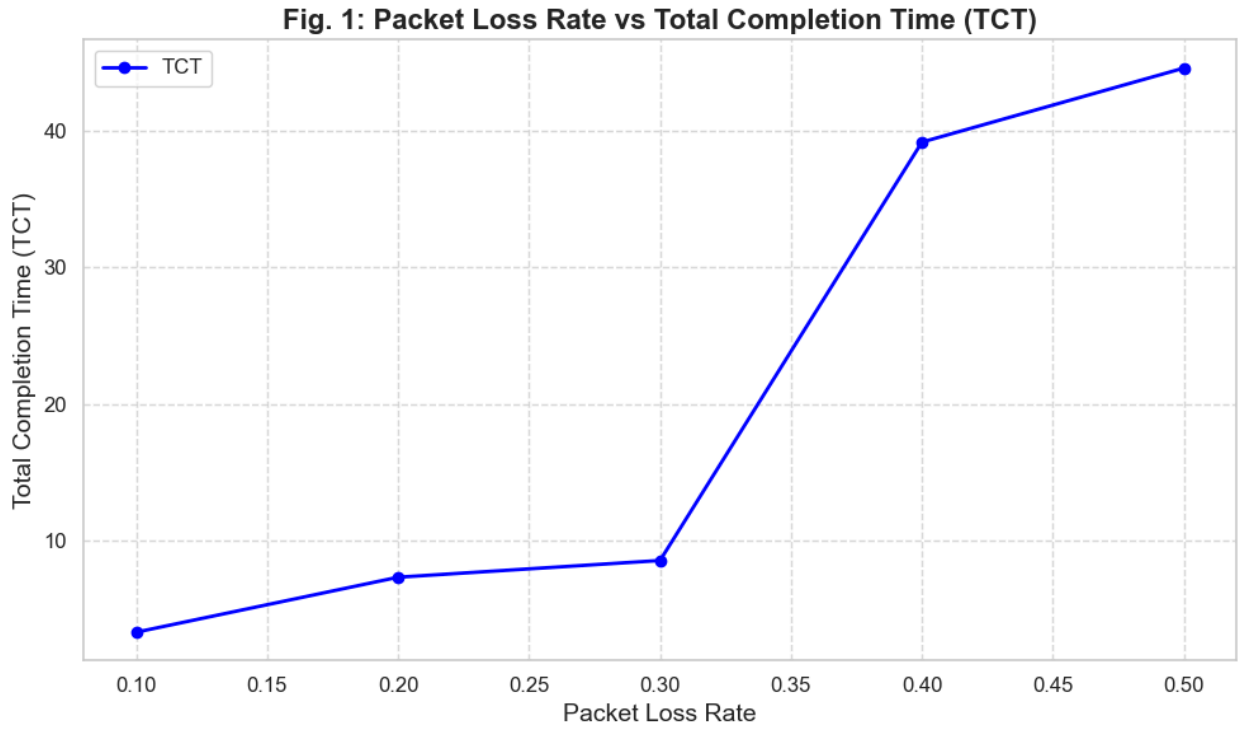
Winning Buyer's Screen:

```
[kbhatia@vclvm177-38 Auction_Socket_Programming]$ python3 auc_client_rdt.py 10.40.129.155 3000 3001 0.2
Connecting to server at 10.40.129.155:3000...
Server: Your role is: [Buyer]

Server: The Auctioneer is still waiting for other Buyer to connect...

Server: Requested number of bidders arrived. Let's start bidding!

Server: Please submit your bid:
Enter bid:110
Server: Bid received. Please wait...

Auction Finished!
You won this item WolfPackSword. Your payment due is $100. Seller's IP: 10.40.130.177

Disconnecting from the Auctioneer Server. Auction is over!
UDP socket opened for RDT
Start receiving file
Msg received: 0
Ack sent: 0
Msg received: 1
Ack sent: 1
Received Data seq 1 : 2000/4518
Msg received: 0
Ack sent: 0
Received Data seq 0 : 4000/4518
Msg received: 1
Ack sent: 1
Received Data seq 1 : 4518/4518
Pkt dropped: 0
Msg received: 1
Ack sent: 0
All data received! Exiting.....
Transmission finished: 4518 bytes / 0.002269 seconds = 1991185.544293 bps
UDP socket closed.
[kbhatia@vclvm177-38 Auction_Socket_Programming]$
```

Performance Measurements:

For Performance measurement we used a File of SIZE=2MB

Figure 1 : Packet loss rate (X axis) and TCT (Y axis)



Values and Trends:

The total completion time (TCT) increases as the packet loss rate increases. This is an expected behavior in our stop and wait protocol, since for every lost packet we have a retransmission, which adds to the overall time to complete the transmission.
In the diagram we can see a non-linear increase, with TCT rising more sharply as the packet loss rate increases.

- At Packet Loss rate: 0.10, TCT is approximately 3.3 seconds
- At Packet Loss rate: 0.20, TCT is rising to approximately 7.3 seconds
- At Packet Loss rate: 0.30, TCT reaches about 8.5 seconds
- At Packet Loss rate: 0.40, TCT increases drastically to 39.1seconds
- And finally at Packet loss rate: 0.50, TCT is at its highest, approximately 44.5 seconds.

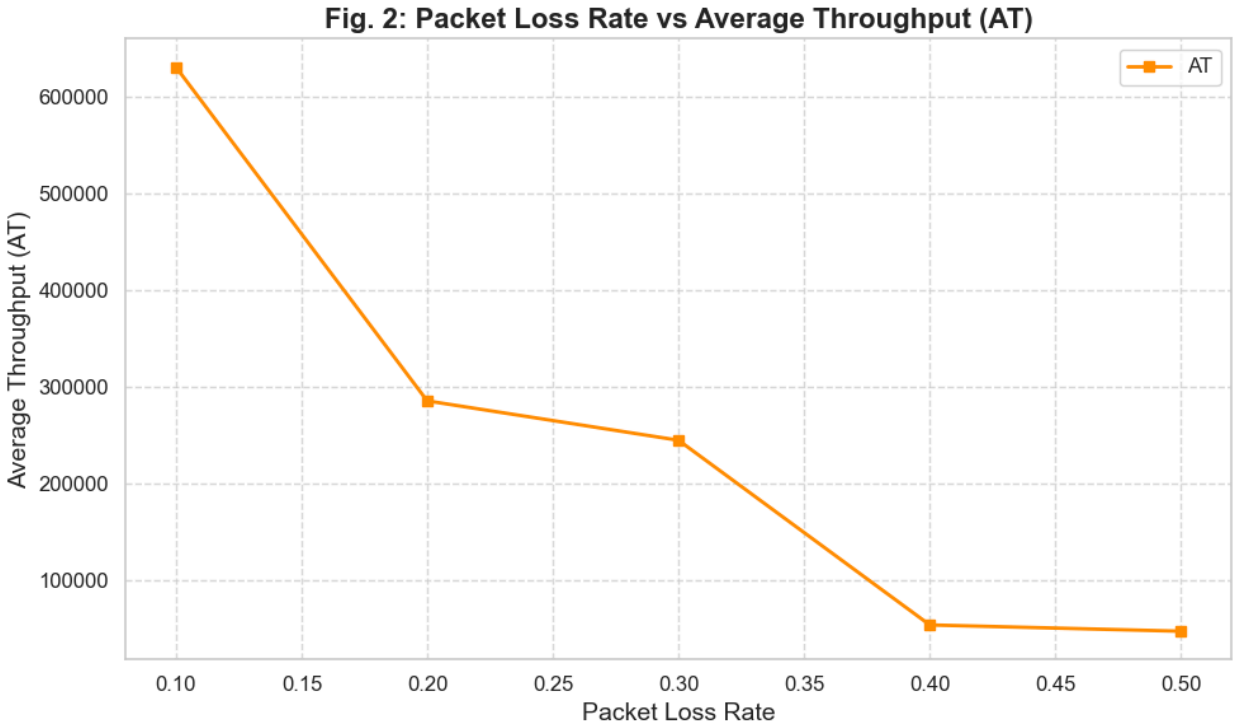
Trends:

1. **Steady increase in TCT:** As the packet loss rate increases from 0.10 to 0.50, the TCT steadily increases, indicating that more time is required to complete the transmission when more packets are lost and need to be retransmitted.
2. **Non linear growth:** The increase in TCT is not linear. The growth is more gradual at lower loss rates and becomes steeper as the packet loss rate increases beyond 0.30.
3. **Small change in higher loss rates:** The change in TCT between packet loss rates of 0.40 and 0.50 is relatively small which suggests that the protocol reaches a point where further increase in packet loss have diminishing impact on TCT.
4. **Timeouts and retransmissions:** In the stop and wait protocol each lost packet triggers a timeout and retransmission, which adds to the total completion time. However, at higher loss rates, most of the time is already spent waiting for retransmissions, so additional losses do not significantly increase TCT.
5. **Saturation Effect:** The protocol reaches a saturation point where further increase in packet loss have less impact on TCT

because the system is already operating inefficiently due to frequent retransmissions.

Conclusion:

The Total Completion Time (TCT) increases as the packet loss rate rises, with more significant growth at higher loss rates due to frequent retransmissions and timeouts in the Stop-and-Wait protocol. However, beyond a certain point (around 0.40), further increases in packet loss have a smaller impact on TCT.



Values and Trends:

The graph shows a declining trend for the average throughput as we keep on increasing the packet loss rate.
The decline in throughput is steepest between a packet loss rate of 0.10 and 0.20

- At a packet loss rate of 0.10 , the approximate AT is 631055.8139 Bps
- At a packet loss rate of 0.20 , the approximate AT is 285486.756455 Bps
- At a packet loss rate of 0.30, the approximate AT is 244739.239041 Bps
- At a packet loss rate of 0.40, the approximate AT is nearly 53545.889908 Bps
- At a packet loss rate of 0.50, the approximate AT is exactly 47043.103494 Bps

Observed Trend:

1. In the stop and wait protocol, each lost packet requires waiting for timeout before retransmission, causing delays in data transfer.
2. As initially the loss rate is low (0 to 0.10) these timeouts are rare, so throughput is high.
3. As soon as we increased the packet loss rate from 0.10 to 0.20 many more packets were lost, leading to frequent timeouts and retransmissions, which caused a sharp drop in the throughput.
4. After reaching a certain level of inefficiency (around packet loss rate = 0.30) further increase in the packet loss rate does not have a dramatic effect on throughput because the protocol is already spending time waiting for timeouts and retransmission of packets.

5. This is due to saturation where additional losses only causes small reduction in throughput.

Conclusion:

There is an inverse relationship between packet loss rate and average throughput: as the packet loss rate increases, average throughput decreases due to more frequent retransmissions and delays caused by lost packets.There was a significant drop in throughput between a packet loss rate of 0.10 and 0.20 where performance degrades sharply. After the sharp drop, the decline in throughput becomes less steep between higher loss rates which suggests that once the protocol reaches a certain level of inefficiency, further increase in packet loss have a diminishing effect on throughput.

How to read Performance.txt

1. We have a performance.txt with the following details:

   pkt_loss_rate,TCT,AT
   0.1,3.323294,631055.8139
   0.2,7.345994,285486.756455
   0.3,8.569055,244739.239041
   0.4,39.166106,53545.889908
   0.5,44.580052,47043.103494

   Order: Packet loss rates, Total completion time, Average throughput
2. Copy the performance.txt to create performance.csv
3. For reproducibility, Run the code create_graph.py