CSC/ECE 573 Internet Protocols, Sections 001
2024 Fall Socket Programming Assignment 2

Team members : Rakesh Kannan(**rkannan3**@ncsu.edu)
Sharmila Reddy Anugula(**sanugul**@ncsu.edu)

How to run the code :

Module Name : auc_server_rdt.py
Description : This script implements an auction server that manages the auction process. It listens for clients (buyers and sellers), handles auction requests, processes bids, and determines the winner based on first-price or second-price auction types.
Usage : Run this script on a server to manage auctions. The first client connection will be treated as a seller, and subsequent connections will be buyers.
Example :
$ python3 auc_server_rdt.py <IP Address> <Port Number>
Where '<IP Address>' - the IP address on which you want the server to listen for incoming connections
'<Port Number>' - the TCP port number on which the server will listen for incoming connections

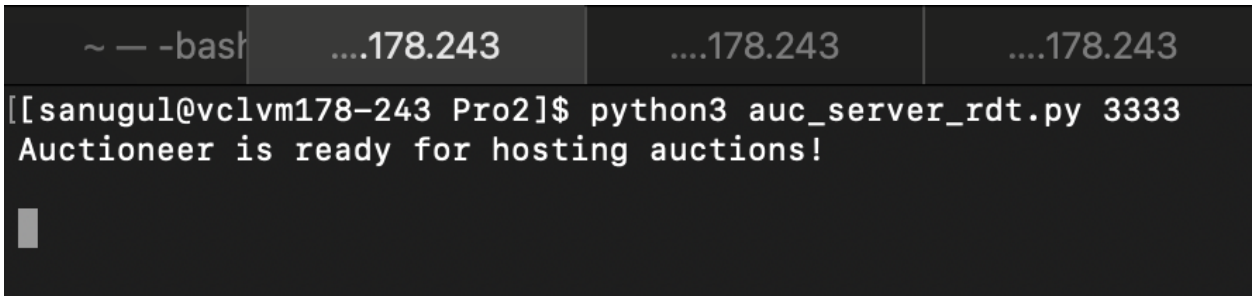
Module Name : auc_client_rdt.py
Description : This script implements a client that can either act as a seller or a buyer. The seller submits an auction request, and buyers submit bids. The server manages the auction process and determines the winner.
Usage : Run this script to connect to the auction server as either a seller or buyer. The server must be running for this client to connect.
Example :
$ python3 auc_client_rdt.py <Server IP Address> <ServerPort> <RDT Port> <rate>
Where '<Server IP Address>' - the IP address of the auction server that the client will connect to
'<ServerPort>' - the TCP port number on which the auction server is listening for incoming connections
'<RDT Port>' - the UDP port number that will be used for the reliable data transfer (RDT) protocol implementation
'<rate>' - the packet loss rate to be simulated for testing the RDT protocol. It's a value between [0.0, 1.0], where: 0.0 means no packet loss
1.0 means 100% packet loss
If we want to run without packet loss then we can set <rate> to 0

Communication between the Seller/Buyer clients and the Auctioneer Server.

1. **Starting the Auctioneer Server:**

The server is started by specifying the TCP port of the welcoming socket.
First run the server i.e., auc_server_rdt.py
$ python3 auc_server_rdt.py <IP Address> <Port Number>
$ python3 auc_server_rdt.py 3333

It starts the service and waits for the client.



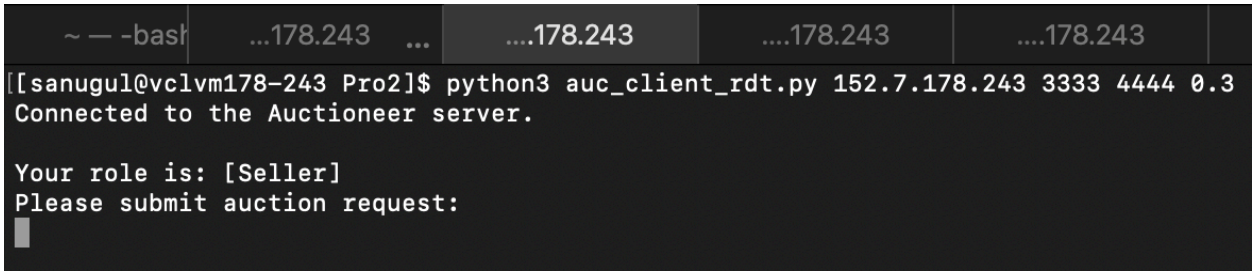
2. **Starting the client (Seller):**

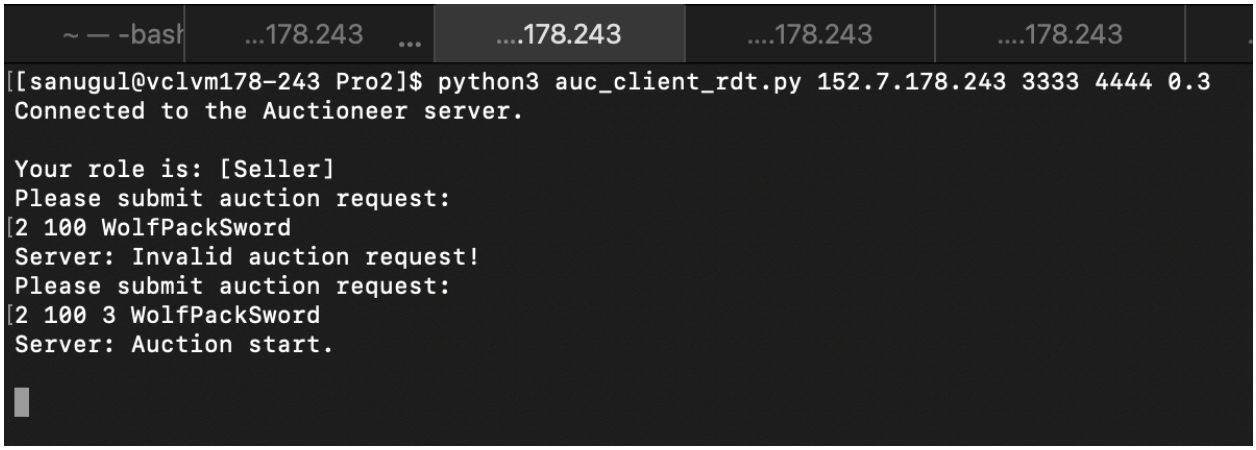
Now run the client i.e., auc_client_rdt.py
$ python3 auc_client_rdt.py <Server IP Address> <ServerPort> <RDT Port> <rate>
$ python3 auc_client_rdt.py 152.7.178.243 3333 4444 0.3

It starts the client and connects to the server. The role of the client (Seller) is received from the server after connection.

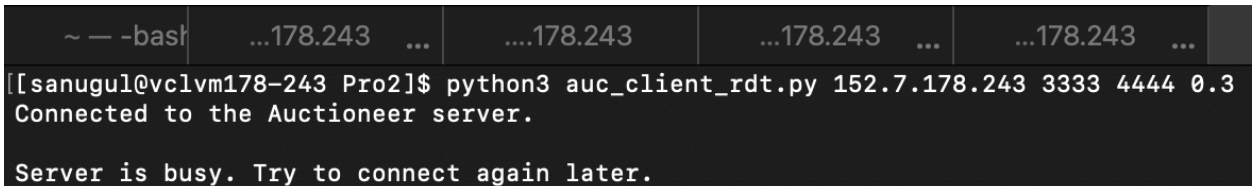
Data validation has been implemented on both the Seller and Buyer sides, and the server will only accept valid requests. A prompt will be issued from the server.
Valid bid (type, min_price, #bids, name).

An additional port number for RDT needs to be provided for the client (Seller).

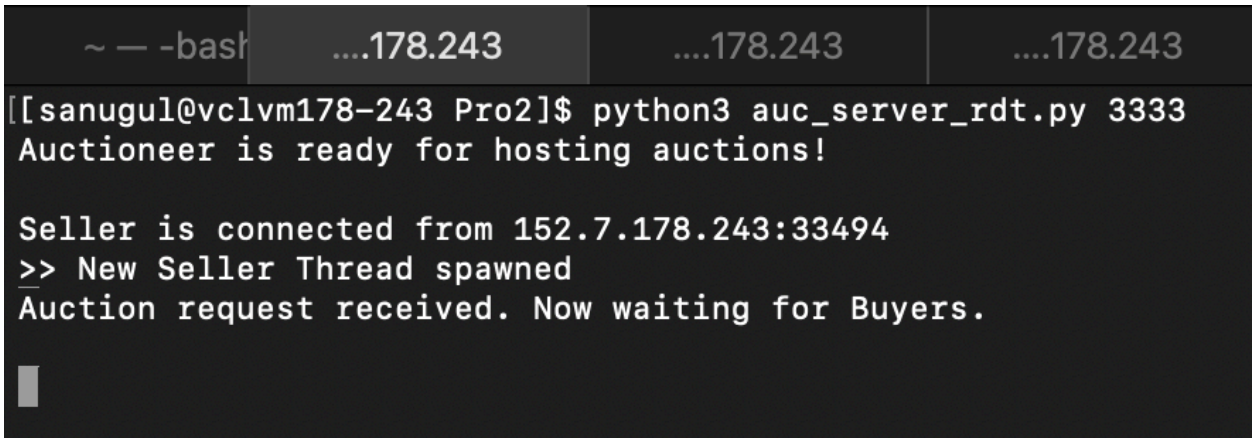




If a client or Buyer tries to connect before the Seller's auction is received, the server will display a "Server Busy" prompt and close the connection.

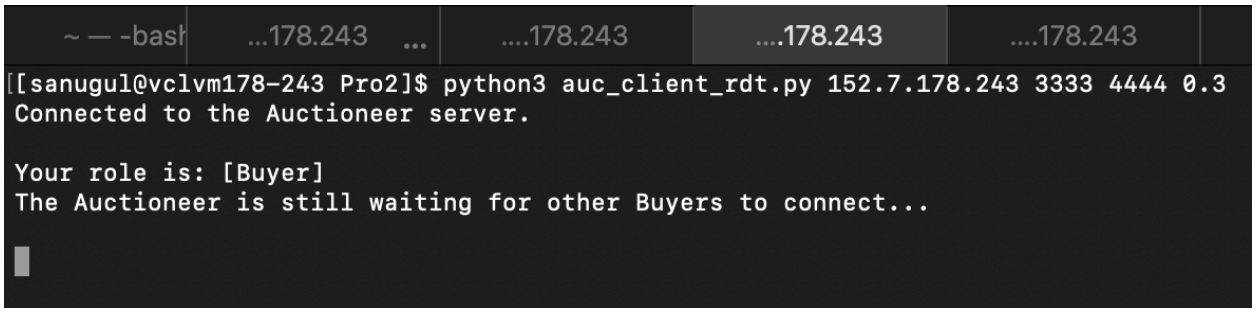


On the server side:
Once the server receives a valid request, it will spawn a new thread and wait for Buyers to connect.

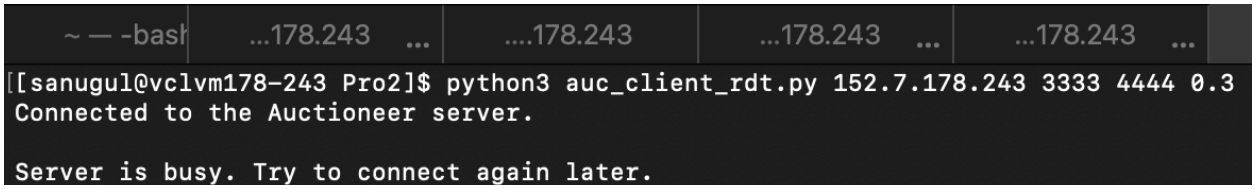


3. **Starting the client (Buyer):**

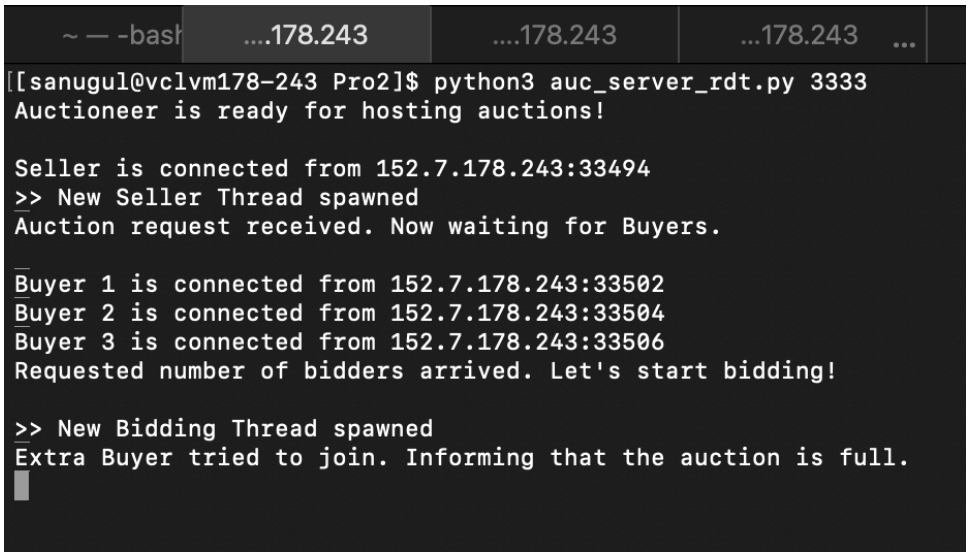
As the server receives a request from the Seller, Buyer clients can start to connect. Once connected, the assigned role was communicated, and a waiting/bid start prompt was automatically sent by the server. The buyer was connected and waiting for other buyers.



The server will only accept the number of buyers specified by the seller. If additional buyers attempt to connect, the client will display a "Server Busy" prompt and close the connection.

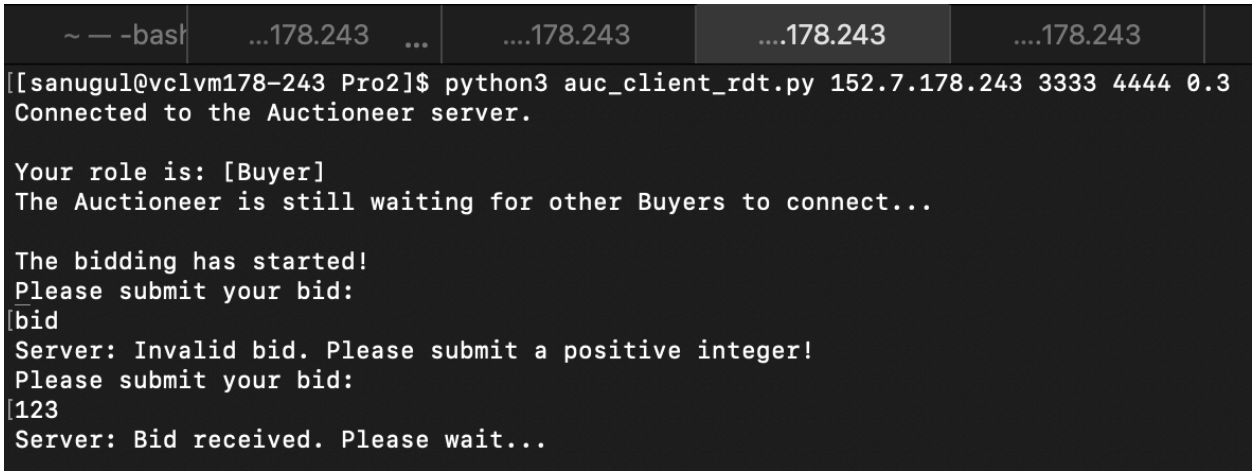


On the server side:
The server will post a message whenever a client connects, inform all clients of their roles, and whether it is still waiting for more Buyers.
Once the required number of buyers are connected, the server will announce the start of bidding, send prompts to all Buyers, and spawn a new thread to handle the bidding process.

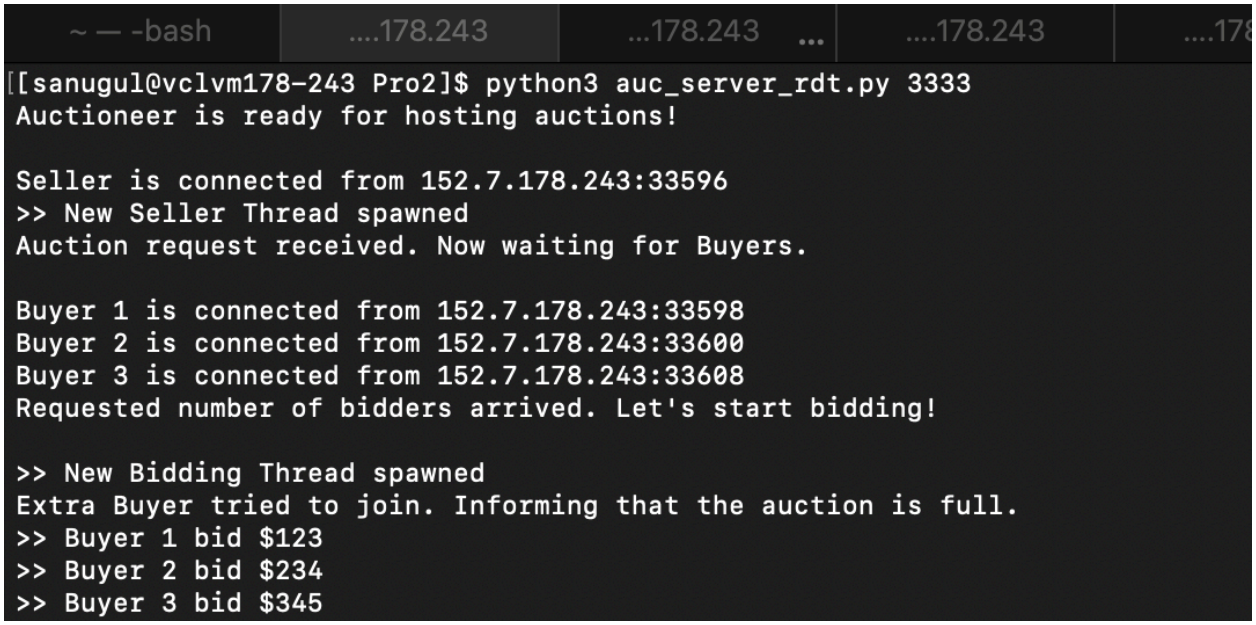


4. **Bidding start :**

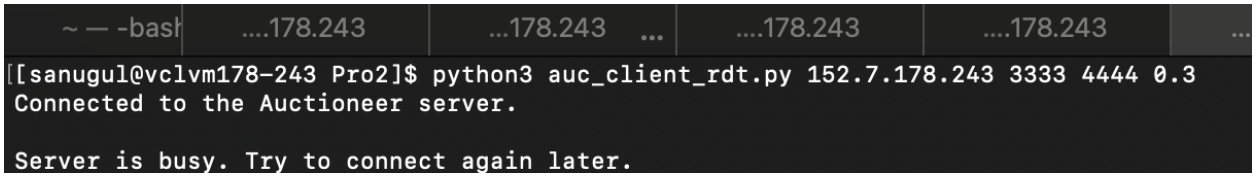
On the Buyer side:
Received the bidding start message from the server. Once the bid is entered, it receives a message confirming that the bid was received and waits for the results.



The server will need to display each bid that is received.



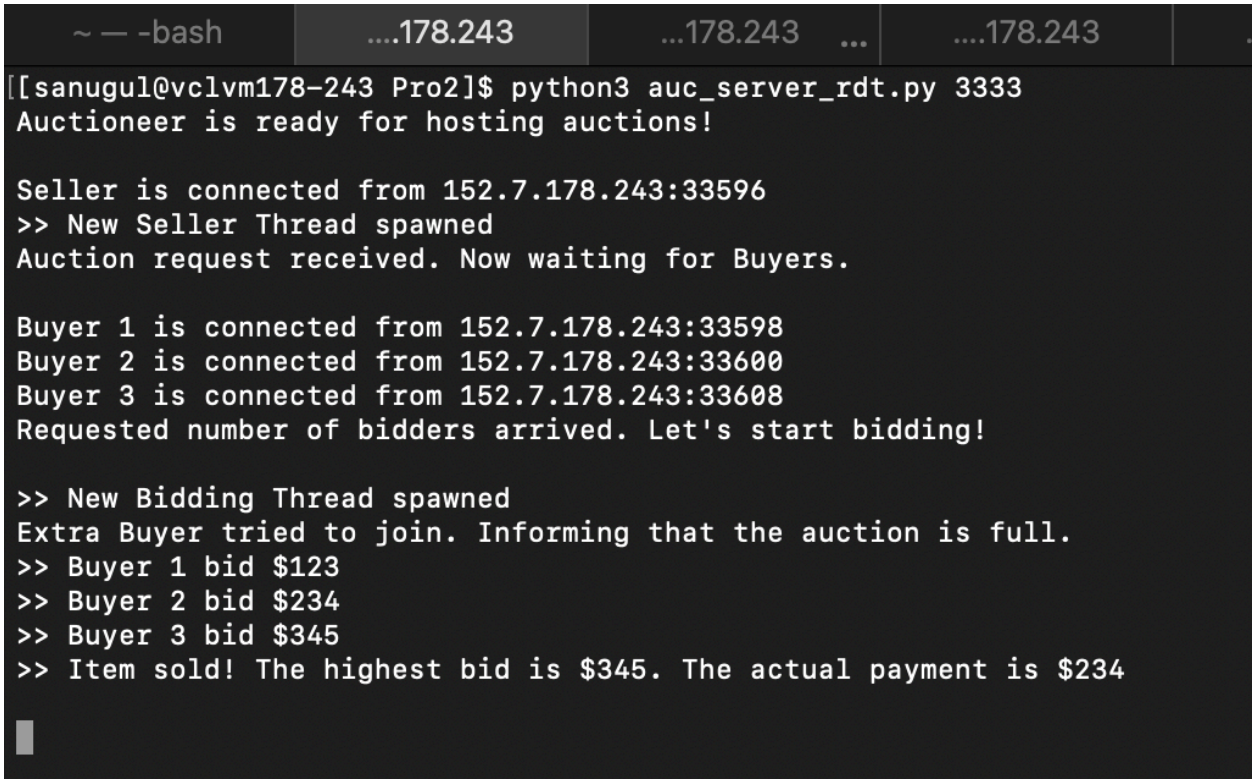
If another client attempts to connect to the server during bidding, the server should accept the connection, inform the client that it is busy, and then close the connection.



5. **Get the final auction result :**

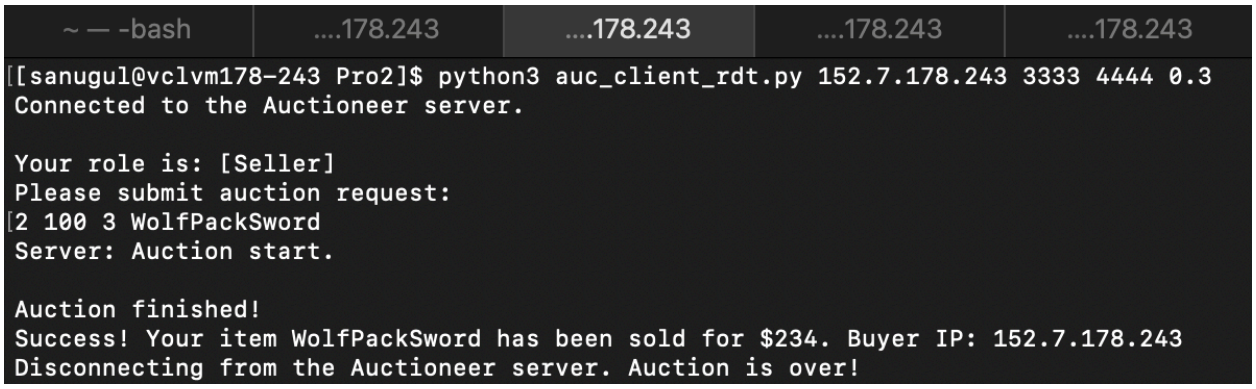
After all bids are received, the server will determine the winning buyer and the price based on the seller's policy defined in the request.

On the server side :



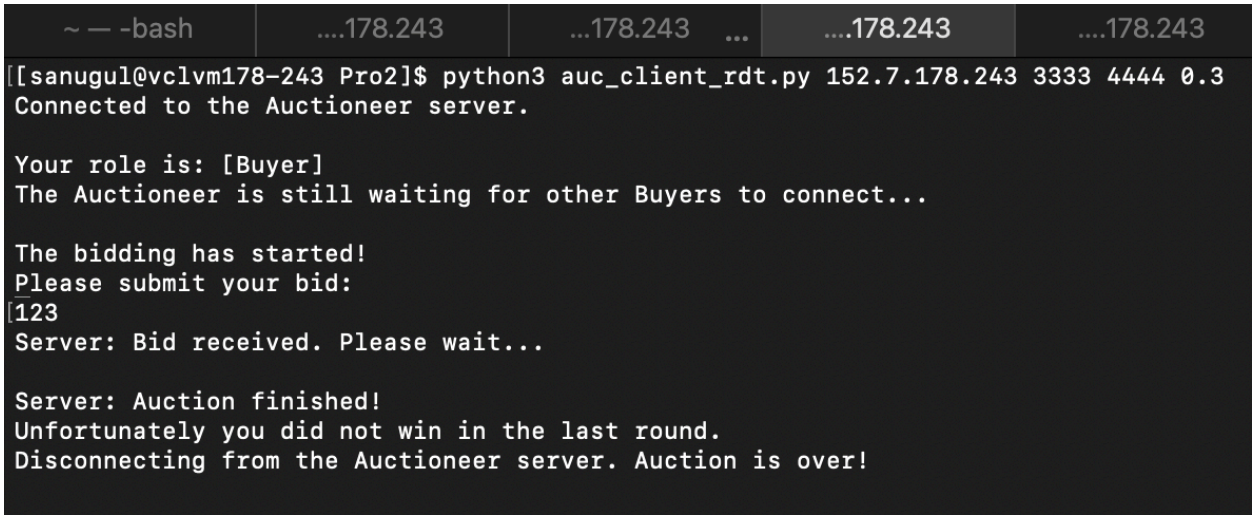
On the Seller side :

The server will then send the results to the seller.

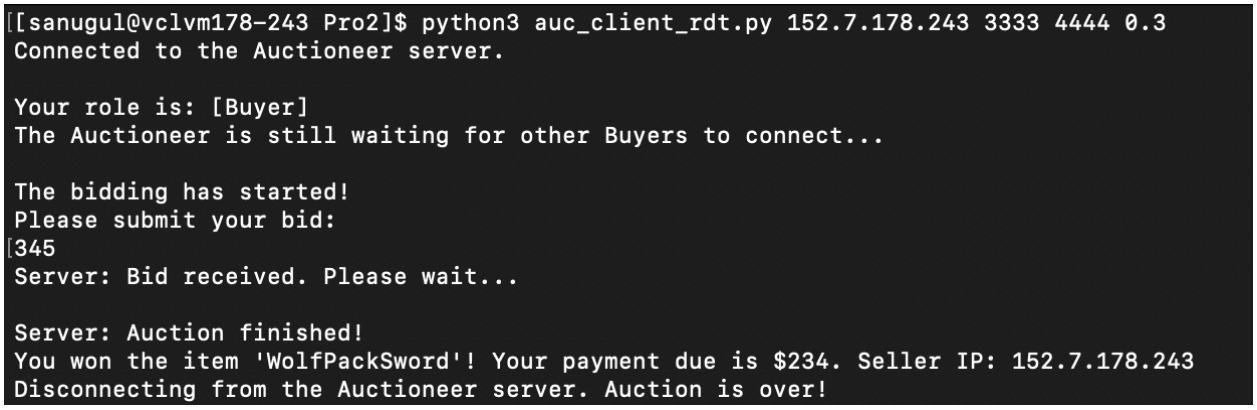


The winning Buyer and the losing Buyer will receive messages from the server. Finally, the server will close connections with all clients. and wait for the next seller to connect for another round of auction.

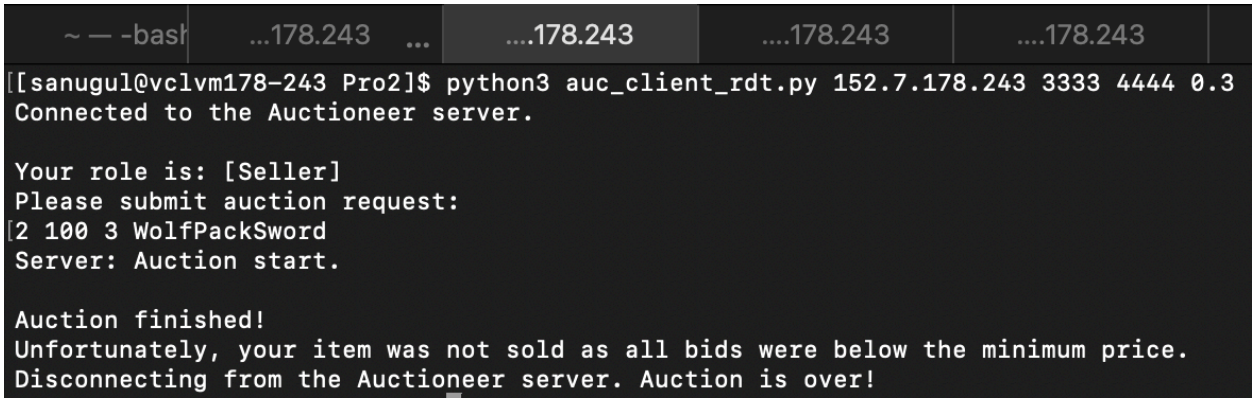
On the losing Buyer side :



On the winning Buyer side :



If the highest bid is below the minimum price, the server notifies the Seller and other clients that the item was not sold, then closes all connections.



After the auction, the seller initiates the Reliable Data Transfer (RDT) protocol to send the file to the Winning Buyer (WB), closing connections with all other buyers. The seller reads the file 'toSend.file', splits it into 2000-byte chunks, and sends them via UDP with sequence numbers. A timer is set when each message is transmitted or retransmitted, with a timeout value of 2 seconds. If the timer times out on a message, the sender retransmits the current message in the buffer.

If packets are lost or arrive out of order, the buyer requests retransmission by sending the same sequence number. Upon receiving the correct packet, the buyer acknowledges it and stores the data. This process continues until the buyer receives a 'fin' control message, signaling the completion of the transfer.

The buyer then writes the received data to 'recved.file'. Throughout the transfer, key status updates, such as the sequence numbers, data sent, and acknowledgments, are output to track the progress of the file transfer.

6. **RDT (no packet loss) :**

With no packet loss, clients need a parameter to specify the no packet loss rate (rate = 0)

$ python3 auc_client_rdt.py <Server IP Address> <ServerPort> <RDT Port> <rate>

$ python3 auc_client_rdt.py 152.7.177.71 4000 3444

On the Seller side :

```
[sanugul@vclvm177-71 2Proj]$ python3 auc_client_rdt.py 152.7.177.71 4000 3444
Connected to the Auctioneer server.

Your role is: [Seller]
Please submit auction request:
2 100 3 Wolf
Server: Auction start.

Auction finished!
Success! Your item Wolf has been sold for $234. Buyer IP: 152.7.177.71
Disconnecting from the Auctioneer server. Auction is over!
UDP socket opened for RDT.
Start sending file.
Sending control seq 0: start 1176996
Ack received: 0
Sending data seq 1: 2000 / 1176996
Ack received: 1
Sending data seq 0: 4000 / 1176996
Ack received: 0
Sending data seq 1: 6000 / 1176996
Ack received: 1
```

On the WB side :

```
[sanugul@vclvm177-71 2Proj]$ python3 auc_client_rdt.py 152.7.177.71 4000 3444
Connected to the Auctioneer server.

Your role is: [Buyer]
The Auctioneer is still waiting for other Buyers to connect...

The bidding has started!
Please submit your bid:
345
Server: Bid received. Please wait...

Server: Auction finished!
You won the item 'Wolf'! Your payment due is $234. Seller IP: 152.7.177.71
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 / 1176996
Msg received: 0
Ack sent: 0
Received data seq 0: 4000 / 1176996
Msg received: 1
Ack sent: 1
Received data seq 1: 6000 / 1176996
```

After sending the complete file, the seller ends the transfer by sending a "fin" control signal. The WB then confirms all data received and outputs the received data bytes divided by the completion time as a statistical result.

On the Seller side :

```
Sending data seq 1: 1174000 / 1176996
Ack received: 1
Sending data seq 0: 1176000 / 1176996
Ack received: 0
Sending data seq 1: 1176996 / 1176996
Ack received: 1
Sending control seq 0: fin
Ack received: 0
UDP socket closed after transfer.
```

On the WB side :

```
Msg received: 1
Ack sent: 1
Received data seq 1: 1174000 / 1176996
Msg received: 0
Ack sent: 0
Received data seq 0: 1176000 / 1176996
Msg received: 1
Ack sent: 1
Received data seq 1: 1176996 / 1176996
Msg received: 0
Ack sent: 0
All data received! Exiting...
Transmission finished: 1176996 bytes / 2.076225 seconds = 4,535,138.440183 bps
UDP socket closed after receiving.
```

7. **RDT (with packet loss) :**

With packet loss, clients need a parameter to specify the packet loss rate

$ python3 auc_client_rdt.py <Server IP Address> <ServerPort> <RDT Port> <rate>

$ python3 auc_client_rdt.py 152.7.177.71 4000 3444 0.4

On the Seller side :

```
[[sanugul@vclvm177-71 2Proj]$ python3 auc_client_rdt.py 152.7.177.71 4000 3444 0.4
Connected to the Auctioneer server.

Your role is: [Seller]
Please submit auction request:
[2 100 3 WolfPackSword
Server: Auction start.

Auction finished!
Success! Your item WolfPackSword has been sold for $234. Buyer IP: 152.7.177.71
Disconnecting from the Auctioneer server. Auction is over!
UDP socket opened for RDT.
Start sending file.
Sending control seq 0: start 1176996
Ack received: 0
Sending data seq 1: 2000 / 1176996
Ack dropped: 1
Msg re-sent: 1
Ack received: 1
```

On the WB side :

```
[[sanugul@vclvm177-71 2Proj]$ python3 auc_client_rdt.py 152.7.177.71 4000 3444 0.4
Connected to the Auctioneer server.

Your role is: [Buyer]
The Auctioneer is still waiting for other Buyers to connect...

The bidding has started!
Please submit your bid:
[345
Server: Bid received. Please wait...

Server: Auction finished!
You won the item 'WolfPackSword'! Your payment due is $234. Seller IP: 152.7.177.71
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 / 1176996
Msg received with mismatched sequence number 1. Expecting 0
Ack re-sent: 1
```

After sending the complete file, the seller ends the transfer by sending a "fin" control signal. The WB then confirms all data received and outputs the received data bytes divided by the completion time as a statistical result.

On the Seller side :

```
Sending data seq 1: 1174000 / 1176996
Msg re-sent: 1
Ack received: 1
Sending data seq 0: 1176000 / 1176996
Ack received: 0
Sending data seq 1: 1176996 / 1176996
Ack received: 1
Sending control seq 0: fin
Msg re-sent: 0
Ack received: 0
UDP socket closed after transfer.
```

On the WB side :

```
Pkt dropped: 1
Msg received: 1
Ack sent: 1
Received data seq 1: 1174000 / 1176996
Msg received: 0
Ack sent: 0
Received data seq 0: 1176000 / 1176996
Msg received: 1
Ack sent: 1
Received data seq 1: 1176996 / 1176996
Pkt dropped: 0
Msg received: 0
Ack sent: 0
All data received! Exiting...
Transmission finished: 1176996 bytes / 1233.400729 seconds = 7,634.151480 bps
UDP socket closed after receiving.
```

8. **File Integrity Check :**

After the transfer, the files are compared for content integrity, and it's ensured that the receiver's file is complete and can be opened.

$ diff received.file tosend.file

```
[[sanugul@vclvm177-71 2Proj]$ diff recved.file tosend.file
[[sanugul@vclvm177-71 2Proj]$ ls -l
total 2344
-rw-r--r--. 1 sanugul NCSU   19235 Nov 20 02:25 auc_client_rdt.py
-rw-r--r--. 1 sanugul NCSU   13446 Nov 20 02:25 auc_server_rdt.py
-rw-r--r--. 1 sanugul NCSU     549 Nov 20 02:25 performance.txt
-rw-r--r--. 1 sanugul NCSU 1176996 Nov 20 02:53 recved.file
-rw-r--r--. 1 sanugul NCSU 1176996 Nov 20 02:25 tosend.file
```

9. **Performance Measurement :**

Metrics when set to a packet loss rate of 0.1

```
All data received! Exiting...
Transmission finished: 1176996 bytes / 122.239493 seconds = 77,028.853432 bps
UDP socket closed after receiving.
```

Metrics when set to a packet loss rate of 0.2

```
All data received! Exiting...
Transmission finished: 1176996 bytes / 398.591531 seconds = 23,623.100997 bps
UDP socket closed after receiving.
```

Metrics when set to a packet loss rate of 0.3

```
All data received! Exiting...
Transmission finished: 1176996 bytes / 704.933373 seconds = 13,357.245315 bps
UDP socket closed after receiving.
```

Metrics when set to a packet loss rate of 0.4

```
All data received! Exiting...
Transmission finished: 1176996 bytes / 1233.400729 seconds = 7,634.151480 bps
UDP socket closed after receiving.
```

Metrics when set to a packet loss rate of 0.5

```
All data received! Exiting...
Transmission finished: 1176996 bytes / 2424.775050 seconds = 3,883.233622 bps
UDP socket closed after receiving.
```

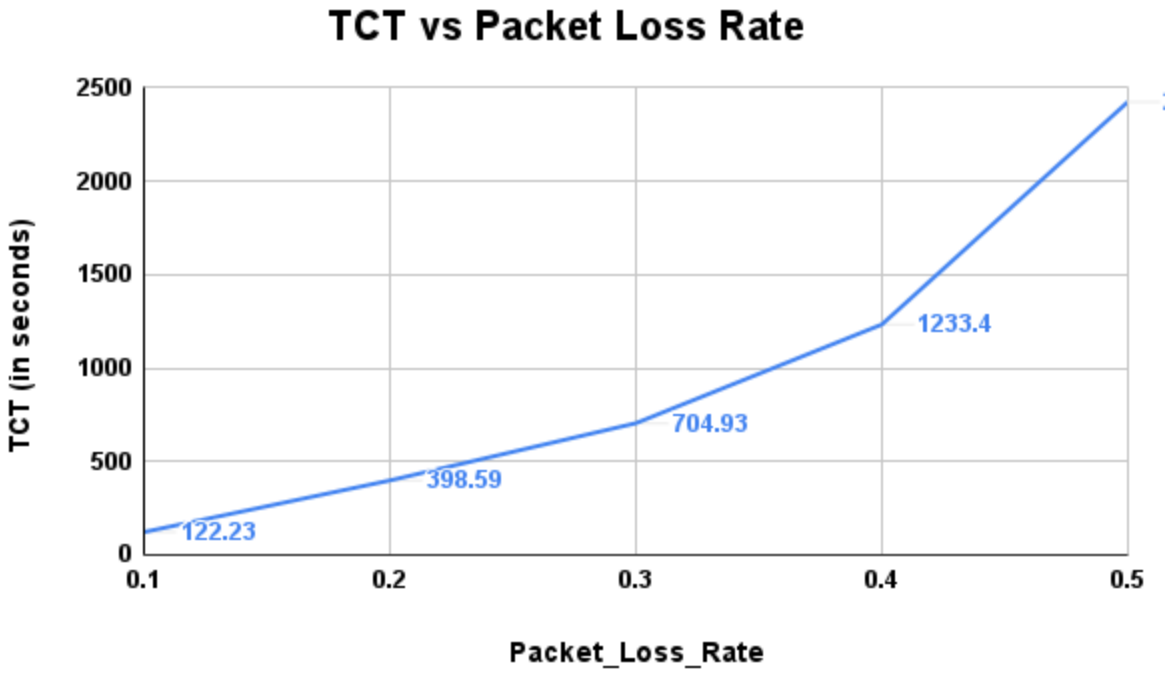


Fig. 1: pkt_loss_rate as the x-axis, and TCT as the y-axis

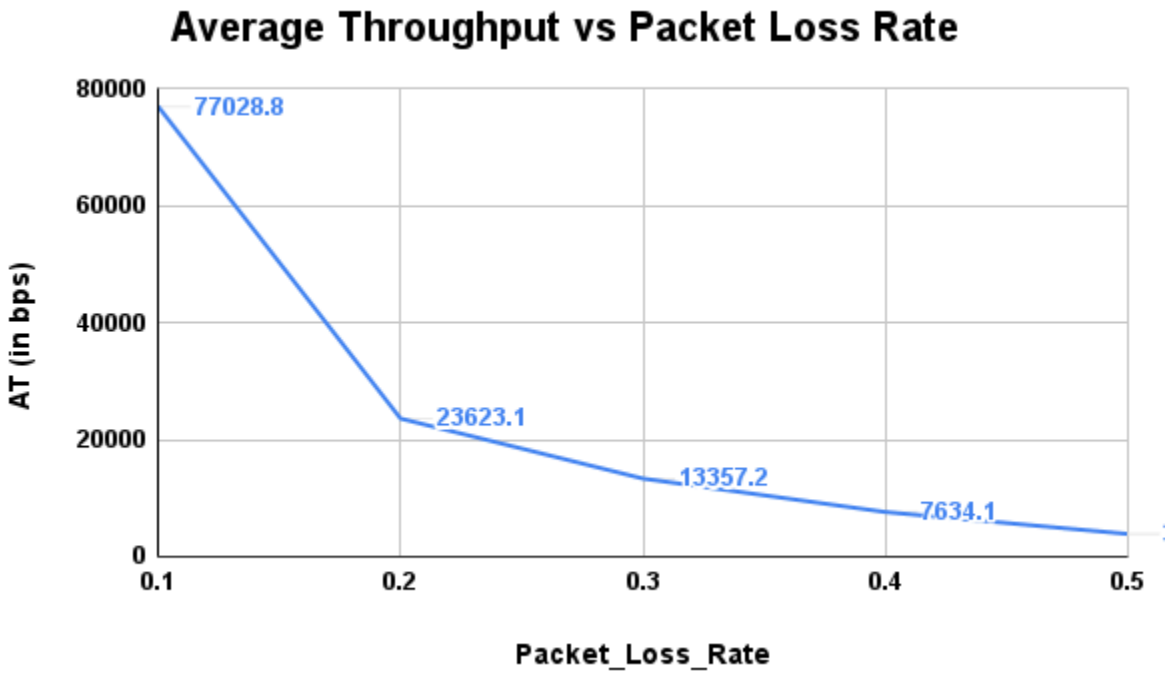


Fig. 2: pkt_loss_rate as the x-axis, and AT as the y-axis

10. Observations :

Throughput Analysis:
The Average Throughput (AT) graph shows a significant downward trend as packet loss rate increases from 0.1 to 0.5:

- At 0.1 packet loss rate, throughput is highest at approximately 77028.8 bps
- Sharp decline to about 23623.1 bps at 0.2 packet loss rate
- Continues to decrease gradually, reaching around 7634.1 bps at 0.4 packet loss rate
- Further drops as it approaches 0.5 packet loss rate

Transfer Completion Time Analysis :
The TCT graph demonstrates an exponential increase as packet loss rate increases:

- Starts at approximately 122.23 seconds at 0.1 packet loss rate
- Rises to 398.59 seconds at 0.2 packet loss rate
- Continues climbing to 704.93 seconds at 0.3 packet loss rate
- Reaches 1233.4 seconds at 0.4 packet loss rate
- Shows steep increase towards 0.5 packet loss rate

Performance Correlation :
The inverse relationship between these metrics clearly demonstrates that:

- Higher packet loss rates severely degrade network performance
- As packets are lost more frequently, the Stop-and-Wait protocol requires more time to complete transfers
- The system becomes increasingly inefficient at higher loss rates, with throughput dropping dramatically while completion times rise exponentially
- The protocol's performance is particularly sensitive to loss rates above 0.2, where the graphs show more dramatic changes in both metrics

11. **Performance.txt :**

Performance File Analysis :
The performance.txt file contains measurements for a network transmission test with the following metrics:

File Structure :

- PLR (Packet Loss Rate): Ranges from 0.1 to 0.5
- Bytes: Constant at 118,725 bytes for all tests
- TCT (Transfer Completion Time): In seconds
- AT (Average Throughput): In bits per second

Key Measurements :
At 0.1 PLR:

- TCT: 16.08 seconds
- AT: 7,381.76 bps
- Best performance point

At 0.3 PLR:

- TCT: 58.14 seconds
- AT: 2,042.19 bps
- Moderate degradation

At 0.5 PLR:

- TCT: 226.31 seconds
- AT: 524.62 bps
- Worst performance point

Graph Correlation :
The data points perfectly align with the graphical representations showing:

- TCT increases exponentially with higher packet loss rates
- AT decreases exponentially as packet loss increases
- The relationship between PLR and performance metrics follows a clear pattern of degradation

The numerical data confirms the visual trends in both graphs, demonstrating the significant impact of packet loss on network performance.