CSC/ECE 573 Internet Protocols, Sections 002

2022 Fall Socket Programming Assignment 2

In this project you will develop a single-client reliable data transfer protocol with UDP, based on the TCP auction server you developed in Project 1. The primary learning objectives are:

- to implement an application-layer rendezvous solution
- to implement a reliable data transfer protocol

1 Requirements

Background

In the last project, we implemented an auction server which sells a certain item on behalf of a seller, and closes the auction with a buyer who submits the highest bid. In this project, we assume that the item being sold by each sender is a digital copy of a large private file (such as an authorized, expensive copy of high-resolution image/video file). For some reason the file can only be sold to one user, and hence only the top-bid buyer should receive this file. The requirement is that whoever wins the bid should be able to receive this file ***reliably***. And in the mean time for some reason the seller decides not to use TCP (perhaps in fear of potential TCP SYN attacks that is too costly to defend against for a small file server).

Our Scenario

On top of your previous auc_server and auc_client programs, you will implement a **UDP-based**, **application-layer reliable data transfer (RDT)** protocol, to transmit the file directly from the seller to the winner of the previous auction. All parties in the previous project are still present, and will complete the auction process as before. However, after the auction, the parties will additionally jointly carry out the transfer of the file from the seller, directly to the winning buyer, with the help from the auction server (who now acts as the index server in a peer-to-peer transfer scenario).

“Auctioneer” Server

For completeness, all procedures from Project 1 are still included (in black font), while new procedures that need to be additionally implemented are marked in blue, same as this prompt.

The server is continuously listening on a welcoming TCP port for clients to connect to it. When the server detects that a new client is trying to connect to it, it creates a new TCP socket to talk to that specific client and does the following:

1. The server runs in two states: “Waiting for Seller” (status 0), and “Waiting for Buyer” (status 1). The server starts in status 0.
2. If the server is running in status 0, and there has not been any client connected to it, the next connected client assumes the Seller role. The server will inform the client to submit an auction request, and wait for it to submit its auction request. For any other in-coming connection before the server receives the auction request, the server replies “Server busy!” to the new client and closes the connection.
    a. To realize this, you can create a new thread to handle the input from the Seller client, while the original thread is still accepting new in-coming clients, sending out a busy message after each connected, and closing the connection.
    b. Display a message when the new thread is spawned.
3. Upon receiving a message from the Seller, the server decodes the message into four fields: <type_of_auction> (1 for first-price, and 2 for second-price), <lowest_price> (positive integer), <number_of_bids> (positive integer < 10), and <item_name> (char array of size <= 255).
    a. If the decoding fails, the server informs the Seller “Invalid auction request!”, and waits for the Seller to send another request.
    b. If the decoding succeeds, the server informs the Seller “Auction request received: [message_received]”. The server then changes to status 1.
4. In status 1, the server will accept in-coming connections from up to <number_of_bids> clients as Buyers. When a client is connected:
    a. If the current number of Buyers is smaller than <number_of_bids>, the server informs the client that the server is currently waiting for other Buyers, and waits for other connections.
    b. If the current number of Buyers is equal to <number_of_bids>, the server tells every connected client “Bidding start!”, and starts the bidding process.
    c. If the current number of Buyers exceeds <number_of_bids>, the server tells the new client “Bidding on-going!”, and closes the connection.
5. The server starts the bidding process by creating a new thread for handling the bidding inputs from Buyers, while the original thread is dedicated to rejecting newly coming clients as in Step 4 during this process. Display a message when the new thread is spawned.
6. During bidding, the server will continuously receive data from any Buyer who has not submitted a bid yet. Upon receiving a message from a Buyer:

a. If the message contains a single positive integer as the bid, the server records the bid with the sending Buyer, tells the Buyer "Bid received. Please wait…", and does not accept any further message from this Buyer.
b. If the message contains any other data, the server informs the Buyer "Invalid bid. Please submit a positive integer!", and attempts to receive another round of message from this Buyer.

7. Upon receiving all bids, the server will run through all Buyers and their bids, and identify the Buyer with the highest bid ***b***. It then decides the result of this auction as follows:
    a. If the highest bid is no less than <lowest_price>, the auction succeeds. The server does the following:
        i. If the <type_of_auction> = 1 (first-price), the server tells the Seller that the item is sold for ***b*** dollars, then tells the highest-bid Buyer that it won in this auction and now has a payment due of ***b*** dollars, and finally tells each of the other Buyers that unfortunately it did not win in this auction. The server then closes the connection to all clients. (We omit the actual payment and good delivery in this project, and defer that to our next project.)
        ii. If the <type_of_auction> = 2 (second-price), the server finds the second highest bid ***b1*** among all bids including the **Seller's <lowest_price>**. The server then tells the Seller that the item is sold for ***b1*** dollars, then tells the highest-bid Buyer that it won in this auction and now has a payment due of ***b1*** dollars, and finally tells each of the other Buyers that unfortunately it did not win in this auction. The server then closes the connection to all clients.
        iii. In either case, the server will additionally send the following information to the seller and the winning buyer:
            1) To Seller: the Winning Buyer's IP address in the format of x.x.x.x
            2) To Winning Buyer: the Seller's IP address in the format of y.y.y.y
    b. If the highest bid is less than <lowest_price>, the server tells the Seller that unfortunately its item was not sold in the auction, and tells all other clients that unfortunately they did not win in the auction. The server then closes all connections.
8. After the auction is done and everyone is informed, the server cleans all the information about this auction including the list of Buyers, and then reset its status to **0**, to wait for any other Seller's connection and auction request.

"Seller/Buyer" Client

When the client starts, it tries to contact the server on a specified port.

For UDP-based message exchange and file transfer, the Seller and the Winning Buyer will implement a **unidirectional**, **Stop-and-Wait** reliable data transfer protocol.

To implement Stop-and-Wait reliable data transfer, each message below is associated with a meta-header containing the following information, all assumed to be 8-bit long unsigned integers: SEQ/ACK (sequence/ACK number), TYPE (message_type). The TYPE takes two

values 0 and 1, where 0 means the message does not contain content of actual file to be transferred, and 1 means the message contains actual content.

These fields precede any actual data in a message. The first message that a party sends via a socket always has sequence number 0. In the following, we omit the meta-header in describing the messages exchanged. We assume the Seller is always the sender (client), while the Winning Buyer is the receiver (server).

Seller Mode

1. If the client receives message from the server to submit an auction request after connection, it holds the role of a Seller.
2. To proceed, the client sends an auction request, in the form of "<type_of_auction> <lowest_price> <number_of_bids> <item_name>", to the server, and waits for server reply.
3. If the server indicates an invalid request, the client must resend a valid request.
4. If the server indicates "Auction request received: [message_sent]", the client waits until the auction finishes to obtain the auction result.

UDP-based reliable message exchange and data transfer

All messages below happen in a separate UDP socket, different from the TCP socket as before. All messages must be authenticated: they must be from the authentic IP address that the server informs, otherwise they must be discarded.

5. If the auction result is success, the Seller client will open up a UDP socket, and start sending messages to the Winning Buyer with RDT. The Seller first needs to read the content of a file "tosend.file" as a **binary byte string**, from its local drive. It then divides the byte string into chunks each with a size of 2000 bytes, representing the maximum segment size.
6. The Seller first sends a transmission start message "start X", where X denotes the total size of the file to be transmitted. This message has type 0 (a control message).
    a. In all message exchange, if any message/ACK is received from an IP address not matching the Winning Buyer's IP, the message/ACK should be directly discarded.
7. After receiving the correct acknowledgement, the Seller starts transmitting the file content.
    a. Seller sends each chunk after receiving the acknowledgement from the Winning Buyer of the previous message exchanged, and put the new chunk into the retransmission buffer.
    b. A timer is set when each message is transmitted / retransmitted, with a timeout value of 2 seconds.
    c. If the timer times out on a message, the sender retransmits the current message in buffer.
8. After all data is sent and acknowledged, Seller informs the Winning Buyer end-of-transmission with a control message "fin", and then exits.

Buyer Mode

1. If the client receives message from the server to wait for other Buyer or start bidding, it holds the role of a Buyer.
2. When the bidding starts, the Buyer sends its bid (a positive integer value) to the server.
3. If the server indicates an invalid bid, the client must resend a valid bid.
4. If the server indicates “Bid received”, the client waits for the final result of the auction.

UDP-based reliable message exchange and data transfer

All messages below happen in a separate UDP socket, different from the TCP socket as before. All messages must be authenticated: they must be from the authentic IP address that the server informs, otherwise they must be discarded.

5. If the auction result is success to the current client (the client is the Winning Buyer -- WB), the client will open up a UDP socket, and then listen on a pre-defined port.
6. WB then waits for messages from the sender.
    a. If an incoming message has unmatched source IP, it should be discarded.
    b. If an incoming message is a data message (with TYPE=1), store the data into a local buffer.
    c. If an incoming message is a control message “start X”, interpret the value of X as the total file size expected to be received.
    d. If an incoming message is a control message “fin”, end the transmission and save all received file content in the local buffer, in-order, to a local file named “recved.file”, and then exit.
    e. Out-of-order / duplicate messages should be properly detected and handled based on RDT, such that they will not stall the transmission process or cause error to the file content.

After the Seller and WB processes end, you should check correctness of your RDT protocol, by manually comparing the files “tosend.file” and “recved.file”. This can be done using the “diff tosend.file recved.file” command on Linux/Mac machines, or the “fc /B tosend.file recved.file” command on Windows machines.

Packet Loss Model

Here, in addition to implementing a faithful RDT protocol, we also want to simulate a network with packet loss (otherwise RDT would not be useful). Without messing up with the underlying layers, we simulate a lossy network by explicitly implement packet loss in the application layer.

Specifically:

- Whenever a UDP message is received at a client (either Seller or WB), before doing anything with the message, client rolls a dice, and directly discard the message with probability <packet_loss_rate>.

- This can be done using a number ways, the easiest being using some existing package such as “numpy”. Specifically, for each message, you can call “numpy.random. binomial(n=1, p=packet_loss_rate)” to generate a flag, which equals to 1 with probability packet_loss_rate, and 0 with probability (1-packet_loss_rate).

The <packet_loss_rate> is given when a client program is started, indicating how lossy the simulated network is. It’s value ranges between [0.0, 1.0].

Performance Measurement

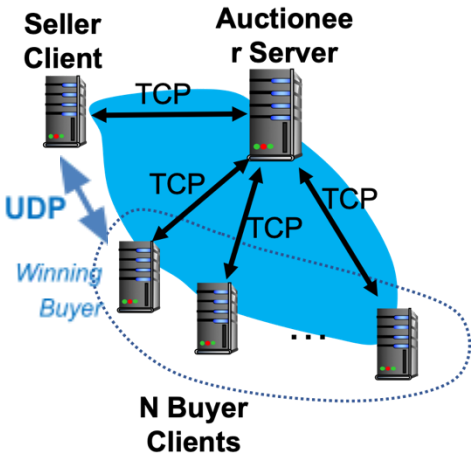
We would like to measure how the implemented Stop-and-Wait protocol perform under different network conditions (packet_loss_rates). To do this, we’ll measure the Transfer Completion Time (TCT) and the Average Throughput (AT) of the given protocol. Both metrics are measured on the Winning Buyer (WB)’s side. Their definitions are as follows:

- Transfer Completion Time (TCT): from the time when the WB sends out the full payment amount, until the time when WB receives the end-of-transmission message.
- Average Throughput (AT): the total number of data bytes received, divided by TCT.

To finish this project, you will need to run the auction for multiple times, each time setting a different <packet_loss_rate> on the participating clients. You will then plot figures about the observed performance metrics (TCT and AT) versus the parameter <packet_loss_rate> that you set. Your eventual goal is to draw conclusion on the performance of Stop-and-Wait when the network becomes more lossy. Guidelines on finishing this part is provided later in this document.

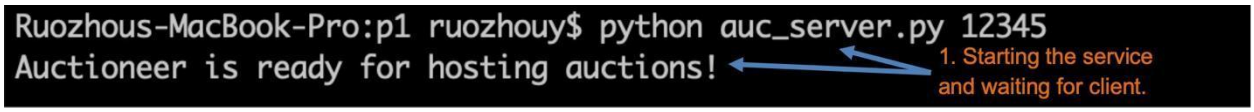
2 Example Output

The figure below represents a visual of the communication between the Seller/Buyer clients and the Auctioneer Server.



2.1 Starting the Auctioneer Server

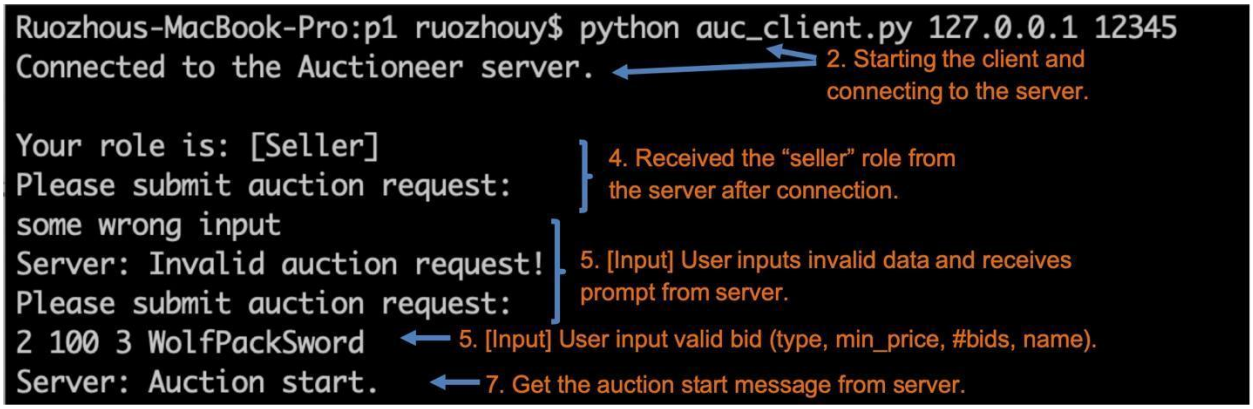
The server is started by specifying the TCP port of the welcoming socket. In the example below, the server is running on port 12345. Display a message as in Step 1 in the figure.



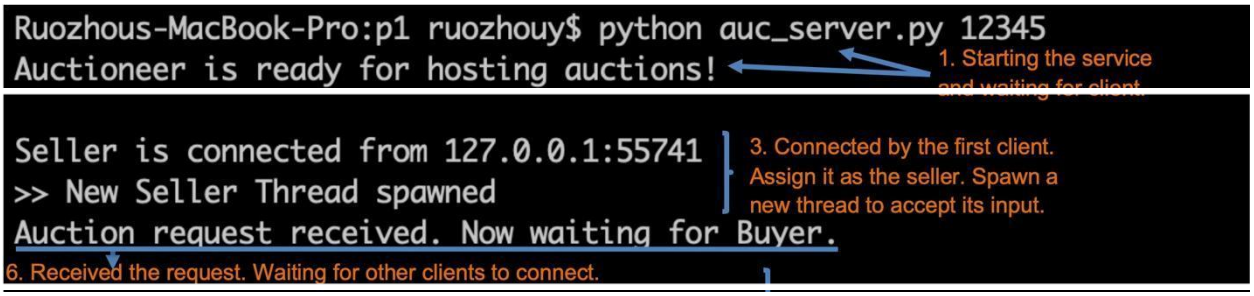
2.2 Starting the client (Seller)

When starting the client, specify the IP address of the Auction Server and the port number where the server is running. In the example below, the server is at IP address 127.0.0.1 running on TCP port 12345. If successfully connected, a message is displayed with relevant information, as in Steps 2 (client) and 3 (server), respectively. Step 4 is displayed when the client receives the Seller role prompt from the server. When a request is entered from the Seller side (Step 5), if it is invalid, the server should return an invalid prompt; if it is valid, the server should send a feedback to the Seller (Step 6) and set itself to wait for Buyers. The Seller displays the a message indicating the auction has started (Step 7). Here the request “2 10 3 WolfPackSword” is used as an example (which means “use auction type 2, starting at $10 minimum, allow 3 bids, for the item name ‘WolfPackSword’ to be sold.”)

On the Seller side:

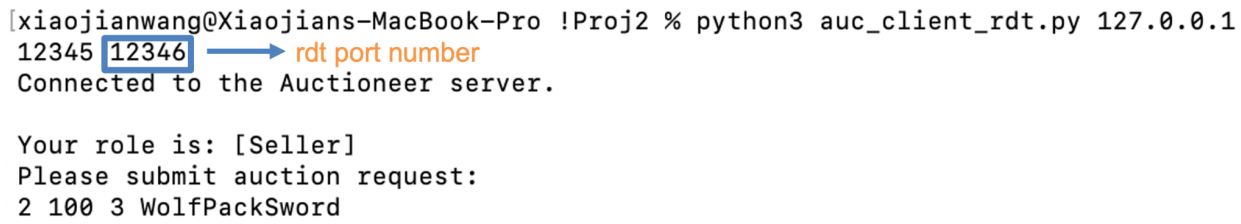


On the server side:



If another client is trying to connect to the server after the Seller is connected but before the request is received, the server will give a busy prompt, and then close the connection:

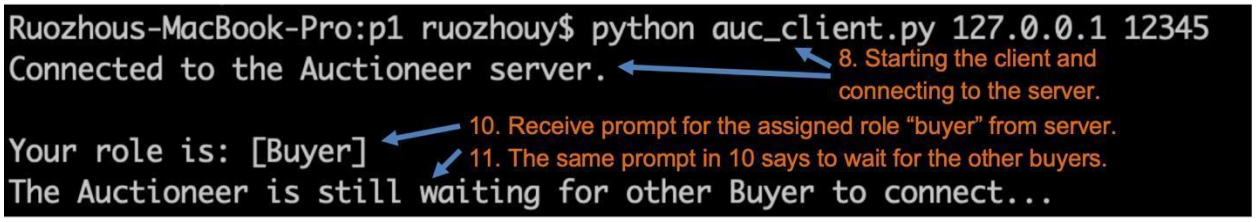
For project 2, an additional port number for rdt needs to be provided for client (Seller).



2.3 Starting the client (Buyer)

After the server receives a request from the Seller, Buyer clients can start to connect (Step 8). When connected, it will be told the assigned role and a waiting / bid start prompt sent automatically by the server after its connection (Steps 10—11).

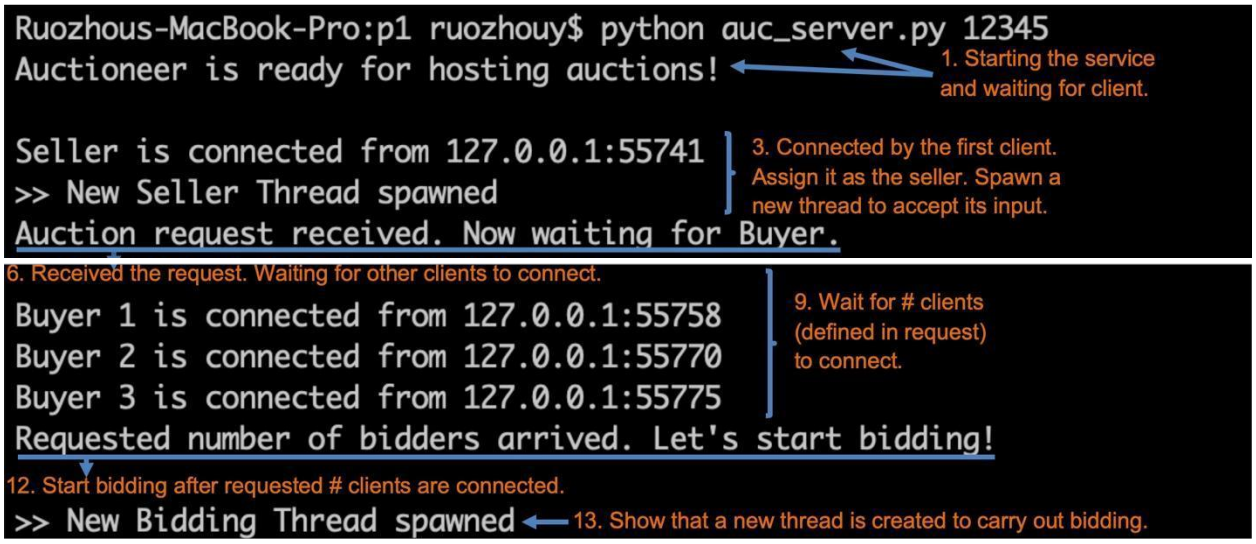
Buyer connected and waiting for other Buyers:



Buyer connected and bidding started immediately afterwards (no waiting prompt and the bidding start prompt immediately follows as in the next subsection):

```
Ruozhous-MacBook-Pro:p1 ruozhouy$ python auc_client.py 127.0.0.1 12345
Connected to the Auctioneer server.

Your role is: [Buyer]
```

On the server side, a message should be posted whenever a client is connected (Step 9), in the meantime all clients should be informed of their roles, and whether the server is still waiting for other Buyers. After the requested number of Buyers are connected, the server will display that the bidding starts (Step 12), send a prompt to all the Buyers to start bidding, and spawn a new thread for handling the bidding (display a message as in Step 13).



For project 2, an additional port number for rdt needs to be provided for clients (Buyer).

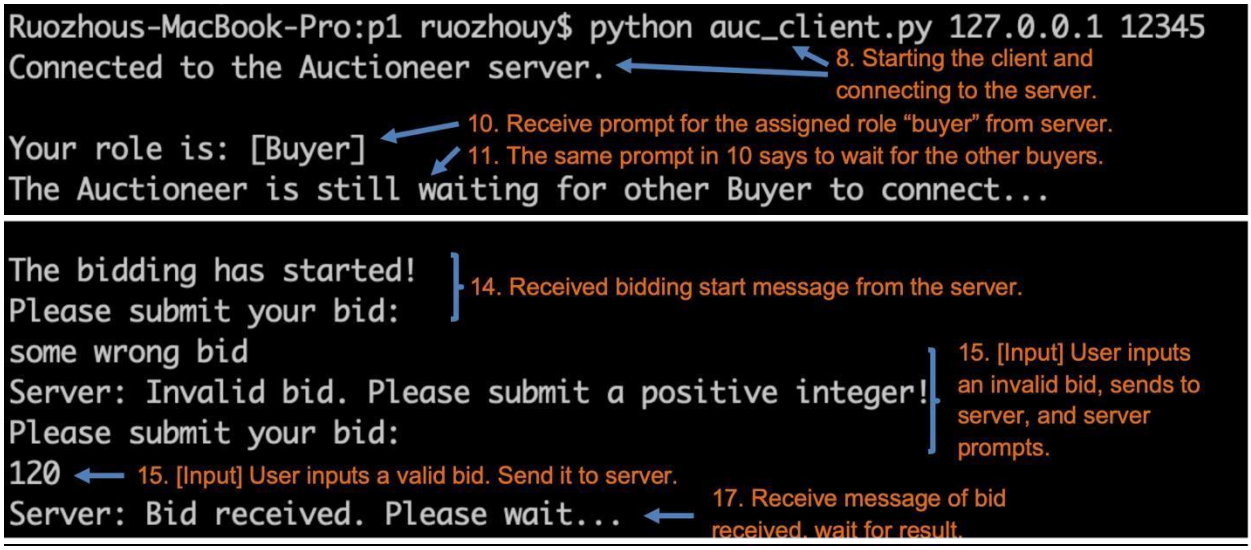
```
[xiaojianwang@Xiaojians-MacBook-Pro !Proj2 % python3 auc_client_rdt.py 127.0.0.1
 12345 12346
 Connected to the Auctioneer server.

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

 The bidding has started!
 Please submit your bid:
```

rdt port number

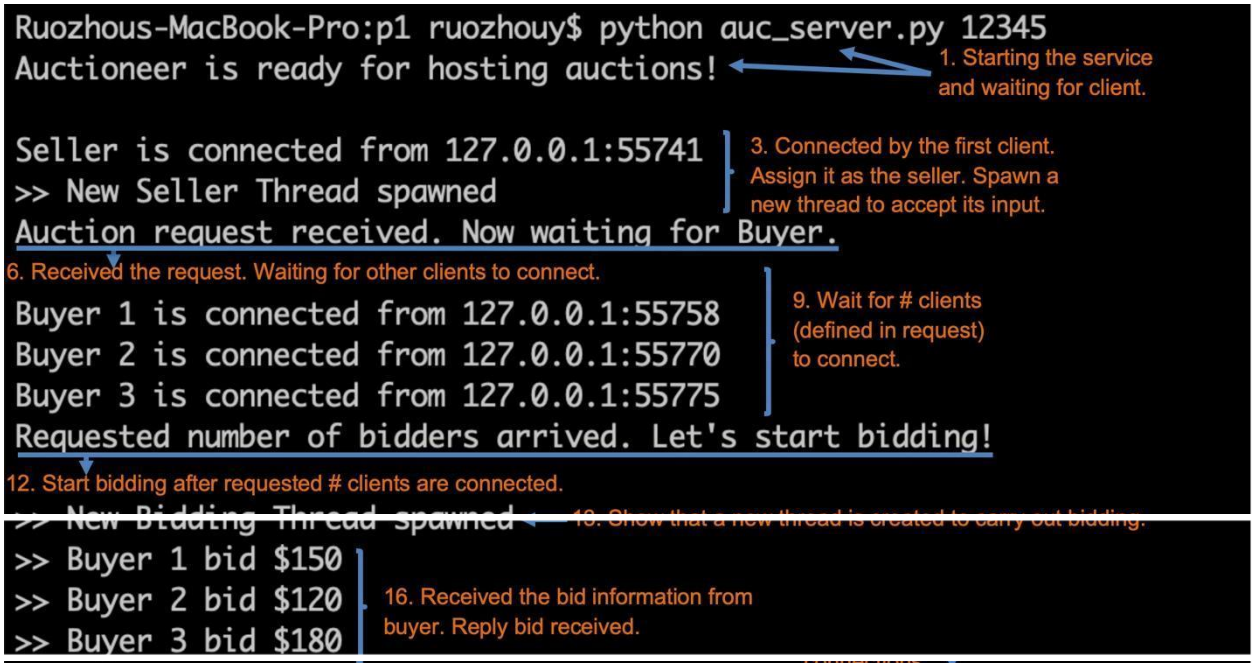
2.4 Bidding start

Once the bidding start, each Buyer will receive a message indicating they can start bidding (Step 14). They will each enter a bid (Step 15), get a prompt from the server indicating the bid is received (Step 17), and wait for the final result.

On the Buyer side:



The Server will need to display each bid received (Step 16).



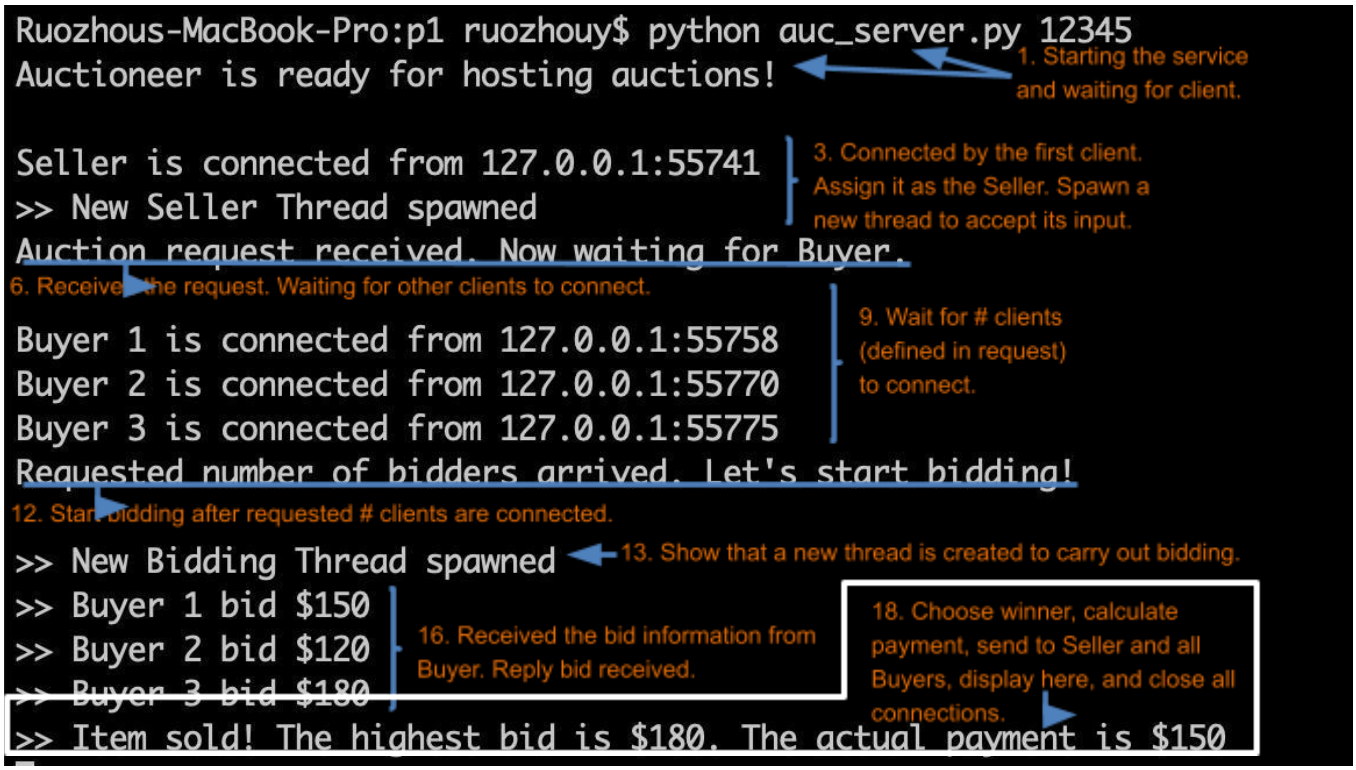
If another client is attempting to connect to the server during bidding, the server should accept this connection, tell the client that it is busy, and close the connection. The client should display a message as follows:

```
Ruozhous-MacBook-Pro:p1 ruozhouy$ python auc_client.py 127.0.0.1 12345
Connected to the Auctioneer server.

Server is busy. Try to connect again later.
```

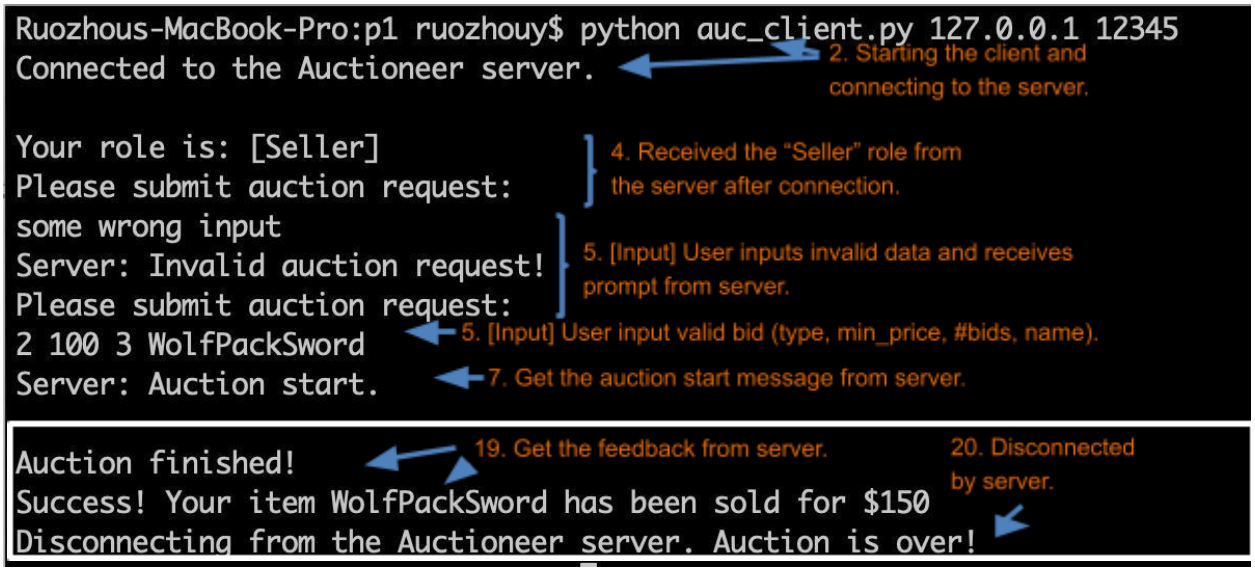
2.5 Get the final auction result

After all bids are received, the server should determine which Buyer wins this auction, and determine the price for the Buyer based on the policy defined in the request by the Seller client (Step 18). The server then needs to send the result via different messages to the Seller, the winning Buyer, and the losing Buyers (Step 19). The server finally closes the connection to all clients (clients display messages as in Step 20), and starts waiting for the next Seller to connect (for another round of auction). The following screenshots are essentially wrap-ups of the entire process of an auction, with highlights on outputs for the current step.

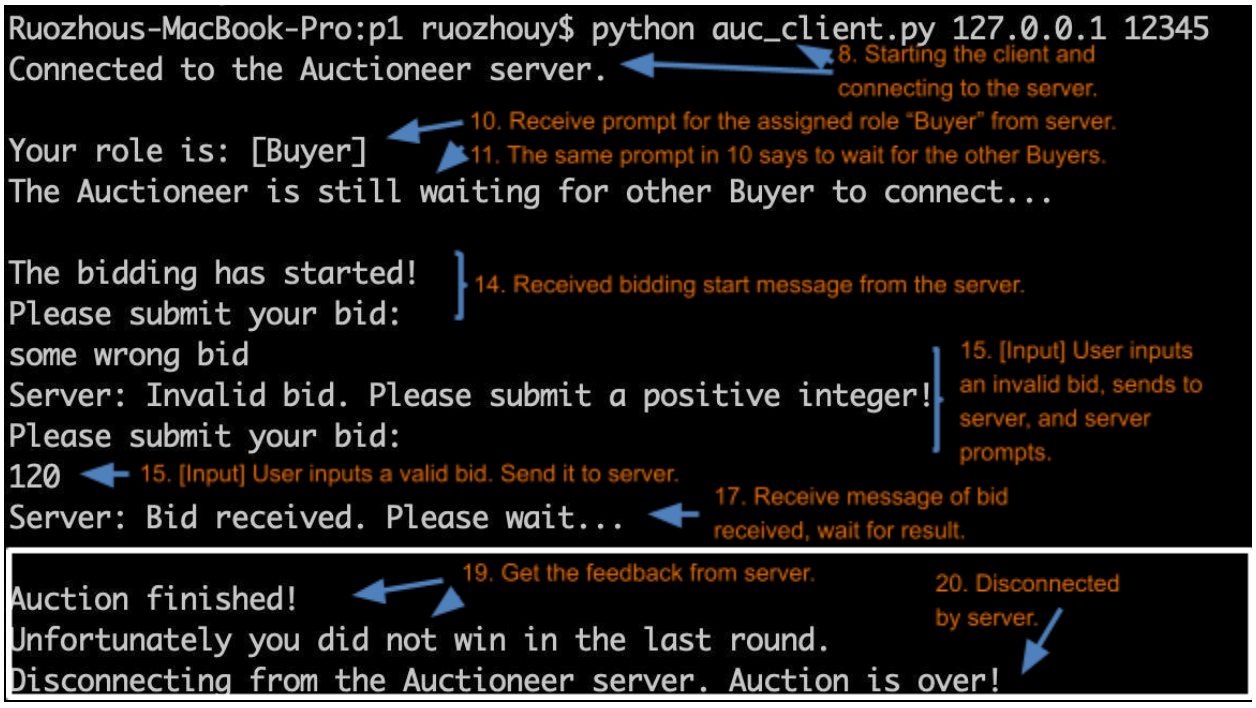
On the server side:



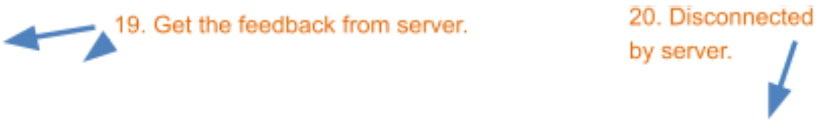
On the Seller side:



On the Buyer side:



Or if this Buyer won:



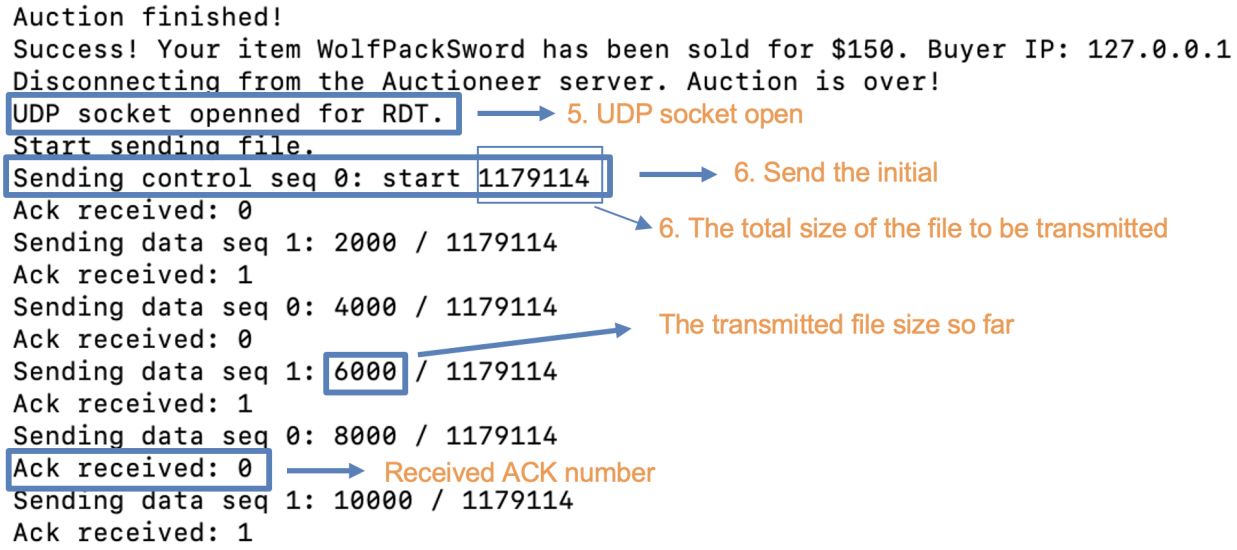
```
Auction finished!
You won this item WolfPackSword! Your payment due is $150
Disconnecting from the Auctioneer server. Auction is over!
```

2.6 RDT (no packet loss)

After the auction is over, the seller starts to initiate the reliable data transfer process to the Winning Buyer (WB), all the other buyers that failed on the auction should be terminated.

After initiation, the seller starts sending the file and the WB starts receiving the file. Some necessary information needs to be explicitly output to indicate the process of file transfer. For example, control seq send, data seq send, Ack received, etc.

On the Seller side:



On the WB side:

```
Auction finished!
You won this item WolfPackSword! Your payment due is $150. Seller IP: 127.0.0.1
Disconnecting from the Auctioneer server. Auction is over!
UDP socket openned for RDT.
Start receiving file.
Msg received: 0
Ack sent: 0
Msg received: 1
Ack sent: 1
Received data seq 1: 2000 / 1179114
Msg received: 0
Ack sent: 0
Received data seq 0: 4000 / 1179114
Msg received: 1
Ack sent: 1
Received data seq 1: 6000 / 1179114
Msg received: 0
Ack sent: 0
Received data seq 0: 8000 / 1179114
Msg received: 1
Ack sent: 1
```

5. UDP socket open

Received message from Seller with sequence number 0

Send out ACK message with number 0

The received file size so far

After the file has been sent out completely, the seller will terminate the rdt with sending control signal fin. The WB will output a message indicating all the data is received and output a statistical result of received data bytes divided by the completion time.

On the Seller side:

```
Sending data seq 0: 1176000 / 1179114
Ack received: 0
Sending data seq 1: 1178000 / 1179114
Ack received: 1
Sending data seq 0: 1179114 / 1179114
Ack received: 0
Sending control seq 1: fin
Ack received: 1
```

8. All data is sent and acknowledged, send a control message fin

On the Buyer side:

```
Received data seq 1: 1178000 / 1179114
Msg received: 0
Ack sent: 0
Received data seq 0: 1179114 / 1179114
Msg received: 1
Ack sent: 1
All data received! Exiting...
Transmission finished: 1179114 bytes / 0.058554 seconds = 161097834.764216 bps
```

Average Throughput (AT)

Total number of data bytes received

Transfer Completion Time (TCT)

2.7 RDT (with packet loss)

With packet loss, the clients need to be provided with a parameter indicating the packet loss rate.

```
xiaojianwang@Xiaojians-MacBook-Pro !Proj2 % python3 auc_client_rdt.py 127.0.0.1
12345 12346 0.5
Connected to the Auctioneer server.
```

After the auction is over, the seller starts to initiate the reliable data transfer process to the Winning Buyer (WB), all the other buyers that failed on the auction should be terminated.

After initiation, the seller starts sending the file and the WB starts receiving the file. Some necessary information needs to be explicitly output to indicate the process of file transfer. For example, the Ack dropped, Msg re-sent, etc.

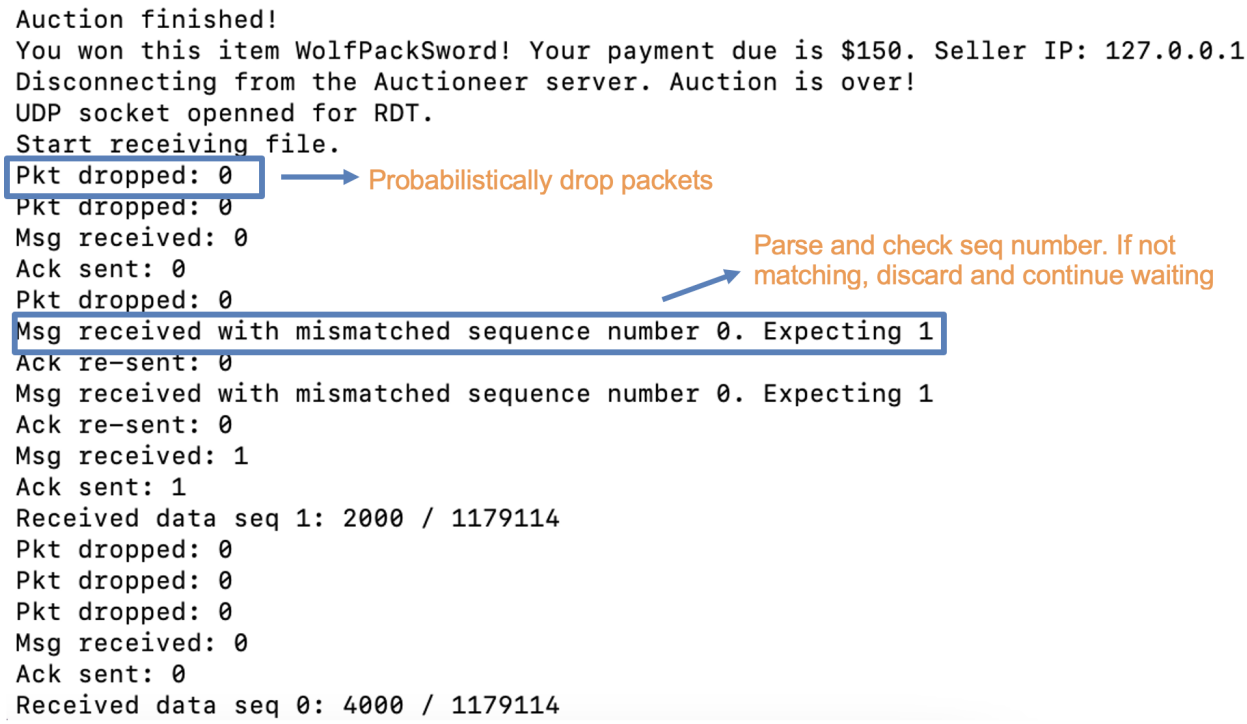
On the Seller side:

```
Auction finished!
Success! Your item WolfPackSword has been sold for $150. Buyer IP: 127.0.0.1
Disconnecting from the Auctioneer server. Auction is over!
UDP socket openned for RDT.
Start sending file.
Sending control seq 0: start 1179114
Msg re-sent: 0
Msg re-sent: 0
Ack dropped: 0
Msg re-sent: 0
Msg re-sent: 0
Ack dropped: 0
Msg re-sent: 0
Ack received: 0
Sending data seq 1: 2000 / 1179114
Ack received: 1
Sending data seq 0: 4000 / 1179114
Msg re-sent: 0
Msg re-sent: 0
Msg re-sent: 0
Ack received: 0
Sending data seq 1: 6000 / 1179114
Msg re-sent: 1
```

7. Timeout and resent message with seq number 0

Probabilistically drop packets

On the Buyer side:



2.8 File Integrity Check

After the final file transfer is complete, make sure that the transferred file at the receiver end is complete and can be opened successfully.

3 Performance Measurement

Your report will need to include performance measurement results of your program given different packet loss rates. These results should be presented in the form of figures, and their raw data should be included along with the report for reproducibility.

To draw the figures, you first need to find a file with file size within **[1, 5]MB**, and then rename it as “tosend.file” and put it in your source code’s directory. You then run the entire process above in VCL for 5 different configurations: pkt_loss_rate = 0.1, 0.2, 0.3, 0.4, 0.5. For each configuration, your client program (Winning Buyer) needs to output 1) the total number of bytes received, 2) the time taken to receive all bytes (TCT), and 3) the average throughput (AT). Based on these data, you will draw two figures:

- Fig. 1: You will use pkt_loss_rate as the x-axis, and TCT as the y-axis.
- Fig. 2: You will use pkt_loss_rate as the x-axis, and AT as the y-axis.

Attach these two figures to your written report. Describe values and trends in these two figures, and draw conclusion on the performance of the implemented Stop-and-Wait protocol given increasing packet loss rates.

4 Implementation Notes

- Write your code in Python 3. The textbook presents sockets programming in Python in Sec 2.7.
  A tutorial on Python is available at http://docs.python.org/tutorial/
  A guide for beginners to learn Python can be found at
  https://developers.google.com/edu/python/
  For network sockets, check out: 4 Beej's Guide to Network Programming
  Python documentation: https://www.python.org/doc/
- Some useful socket functions: **.send(), .sendto(), .recv(), .recvfrom(), .close(), .accept(), .bind(), .listen()**
- For starting new threads, you might want to use **thread.start_new_thread()**
- For generating random numbers for packet loss, a naïve way is to directly use the **random.random()** function (**import random** first), which generates a value in [0, 1]. Comparing the generated value to a threshold (such as pkt_loss_rate) will generate a Bernoulli random variable.
  - Alternatively, packages such as **numpy.random** directly provide Bernoulli random variables, such as the **numpy.random.binomial** function. You need to install numpy before importing and using it.
- Include proper error checking, when the server and a client communicate.
- During development, you can use your own system to write and debug your program. You will of course need to install and setup Python 3.
- You can run the server and the clients locally. The server would be running locally on your system on a specified port. Therefore, the clients will try to connect to 127.0.0.1 on that port.
- Test your code using VCL systems, see Section 5.

5 Grading done on VCL systems

To grade your assignment, we will use NCSU's Virtual Computing Lab (VCL) machines. So once you are done testing locally your code, then you should try it out on VCL. Go to **vcl.ncsu.edu** and make a reservation for the VCL image called **"CSC/ECE573-002 F2022"**. This image is for Linux CentOS 7 and already has Python 3.6.8 installed.

When you are testing your code on the VCL systems, you have to run your Auctioneer Server on ports in the range **[3000 to 5000].** Other ports are blocked via firewall rules so these are the only ports that will work. Your code should handle an arbitrary number N of clients connecting, and you should test it for at least N=3 on VCL. So you can request and reserve **up to five VCL Virtual Machines (VMs):** one VM for Auctioneer Server, one VM for Seller Client, and up to three VMs for Buyer Clients. (see first figure in Section 2).

When grading, the TA will run your code on VCL, with a specific “tosend.file” for grading. This requires that your programs need to work for any file, not just for the file that you picked for testing. To ensure error-freeness, you should test your code on multiple “tosend.file”s to make sure it works properly.

Since this project is based on project 1, in case you didn't get the full credit of project 1 and find it is difficult to continue working for project 2 based on your project 1 (mainly the part where the auction server sends IP addresses of both parties to Seller and WB), you can manually exchange IP addresses for two client programs to implement RDT between them (or implement a simple index server just to exchange the IP addresses of the two clients). You will lose up to 5 points for doing so.

Instruction of file transfer

You can use scp (for MacOS or Linux) or winscp (for Windows) to transfer files to the VCL machines.

- scp: if you want to transfer your local file to remote machine, the following command can be used:
  scp your_local_path_and_filename username@remote_ip_address:remote_path_stores_this_file
  If you want to transfer a directory to the remote machine, add the ‘-r’ option after the ‘scp’ command.
- winscp: Once you create a new session with SFTP protocol and login in that machine, you can directly transfer local files to the remote machine by drag and drop.

6 Submission Details

The project is due Tue, Nov 15, 2022 at 14:55 p.m. Anything beyond the deadline receives a 0% score.

6.1 Team Work

You will be able (but not required) to work in teams of two. If you choose to work with a partner, there won’t be any excuses in regard to partner not doing her/his share of the work.

Everyone in the team needs to submit the final version of the project. Please make sure that each submission includes the names of all the members of the team.

6.2 Plagiarism

The work submitted has to be yours. If you do end up using some reference code you found online, make sure that you cite the website. Make sure you comment all important lines so that it is clear you have a good understanding of the code.

6.3 What to submit

1. Well-commented **source files** (in python): **auc_server_rdt.py** and **auc_client_rdt.py**.

Put your name(s) at the top of each source file.

2. Performance measurement data: **performance.txt**, which contains the number of bytes, TCT and AT values for each of your experiments, which should match the figures in your written report. How the content of performance.txt should be interpreted should be clearly explained in the README.pdf

3. A **README.pdf** file. In the README file, please provide the following information:
    - Your name(s)
    - Your unityid(s)
    - **Screenshots** of you testing your code, i.e. running an auction server, connecting via clients, etc. Add discussion on what the different screenshots mean.
    - **A detailed description on how to compile & run your code, including any package required and how to install.**
    - Performance measurement figures, explanations and conclusions.
    - How to interpret the performance.txt file.
    - (Optional) Any other thing you want to tell us.

7 Grading Rubric

This is how your project will be graded:

Points	Description
8	**README Format:** It is well formatted with all required details
9	**Output:** Terminal output similar to the skeleton
4	Seller and WB initiate the reliable data transfer process and can communicate with each other
4	UDP socket created
5	Server distributes the IP address of Seller and WB correctly
4	Seller starts the file transfer sending process correctly
4	WB starts the file receiving process correctly
7	Without packet loss, the Seller sends out the file correctly and gets the feedback from WB correctly
7	Without packet loss, the WB receives the file correctly and gets the feedback from Seller correctly: get the message from Seller, send out Ack.
4	Timeout be handled correctly
7	With packet loss, the Seller responses to the packet drop correctly
7	With packet loss, the WB responses to the packet drop correctly
3	Seller terminates the sending process when file transfer completes

5	WB receives the whole file at the end of file transfer and the file can be opened successfully
4	WB displays the 1) number of bytes received, 2) the time taken to receive all bytes (TCT), and 3) the average throughput (AT).
5	Performance measurement figure: pkt_loss_rate vs TCT
5	Performance measurement figure: pkt_loss_rate vs AT
4	Explanations and conclusions based on the measurement figure
4	**Source code:** Well commented
Possible Extra Credit or Deductions	
-2	Name(s), date, etc. not included at the top of each .py file
-5	Incorrectly Named Files (**auc_server_rdt.py and auc_client_rdt.py**)
-5	Additional, irrelevant output displayed to the Console Window
-10	Code results in an error
-100	Submitted after 14:55 PM on 11/15/2022