COEN 366 Project

P2P Auction System

Prepared by
Scott McDonald 40211626
Mohammed Zahed 40116292
Jaskirat Kaur 40138320

Submitted on
April 12th, 2024

1. Introduction

The purpose of this project was to deepen our understanding of various communication protocols and networking programming. This was accomplished through the development of a Peer-to-Peer auction system. This system would permit buyers and sellers to interact through a centralized server, where they could send requests through either UDP or TCP communication protocols.

This system was developed in Python, using its native socket library to communicate between the clients and the server.

This report contains the following sections:

- **System Assumptions**
- **System Design and Analysis**
- **Implementation**
- **Testing and Debugging**
- **Conclusion**

2. Assumptions

The system description states that clients must register to the system before performing any of the other actions. Our team assumed that clients would want to access their account in different sessions, therefore a login system was implemented to retrieve their previous session data. This in effect provided improved error handling in the event of a failure on the client's end, since they would not lose access to any of their previous bids.

3. System Design & Analysis

The System takes into consideration that the server and client are running on different devices. So they do not share any common configuration file in between them. The system is implemented as a client-server architecture using both UDP and TCP protocols. The server manages user registration, login, item listings, subscriptions to item updates, bidding, and auction finalization. UDP is primarily utilized for handling non-critical communication such as registration requests, bidding updates, auction announcements, and subscription management due to its speed and simplicity. The server maintains persistent records by saving the auction state, user information, and active subscriptions in a JSON file, ensuring continuity across server restarts.

The client application facilitates user interaction by providing an interface for registration, login, item bidding, auction listing, and subscribing to auction notifications. It maintains dedicated listener threads for both UDP and TCP communications, ensuring responsiveness to server messages. The system is carefully designed to handle concurrency, employing threading and synchronization mechanisms such as locks, ensuring thread-safe operations.

Category	Message Name	Message Format	Direction	Description
Registration	REGISTER	REGISTER RQ# Name Role IP UDP_Port TCP_Port	Client → Server	Register a peer with name, role, and contact info
	REGISTERED	REGISTERED RQ#	Server → Client	Confirmation of successful registration
	REGISTER-DENIED	REGISTER-DENIED RQ# Reason	Server → Client	Denial of registration with reason
	DE-REGISTER	DE-REGISTER RQ# Name	Client → Server	Client requests deregistration
Item Listing	LIST_ITEM	LIST_ITEM RQ#	Seller → Server	Seller lists an item for auction

Category	Message Name	Message Format	Direction	Description
		Item_Name Item_Desc Start_Price Duration		
	ITEM_LISTED	ITEM_LISTED RQ#	Server → Seller	Server confirms item listing
	LIST-DENIED	LIST-DENIED RQ# Reason	Server → Seller	Server denies item listing
Subscriptions	SUBSCRIBE	SUBSCRIBE RQ# Item_Name	Buyer → Server	Subscribe to auction updates for an item
	SUBSCRIBED	SUBSCRIBED RQ#	Server → Buyer	Server confirms subscription
	SUBSCRIPTION-DENIED	SUBSCRIPTION-DENIED RQ# Reason	Server → Buyer	Subscription is denied with reason
	DE-SUBSCRIBE	DE-SUBSCRIBE RQ# Item_Name	Buyer → Server	Buyer unsubscribes from item updates
Auction Announcement	AUCTION_ANNOUNCE	AUCTION_ANNOUNCE RQ# Item_Name Desc Current_Price Time_Left	Server → Buyer	Notify buyers of auction updates
Bidding	BID	BID RQ# Item_Name Bid_Amount	Buyer → Server	Buyer places a bid
	BID_ACCEPTED	BID_ACCEPTED RQ#	Server → Buyer	Bid accepted
	BID_REJECTED	BID_REJECTED RQ# Reason	Server → Buyer	Bid rejected with reason

Category	Message Name	Message Format	Direction	Description
Bid Updates	BID_UPDATE	BID_UPDATE RQ# Item_Name Highest_Bid Bidder_Name Time_Left	Server → All Subscribed users.	Notify all participants of new highest bid
Negotiation	NEGOTIATE_REQ	NEGOTIATE_REQ RQ# Item_Name Current_Price Time_Left	Server → Seller	Invite seller to reduce price if no bids received
	ACCEPT	ACCEPT RQ# Item_Name New_Price	Seller → Server	Seller agrees to lower price
	REFUSE	REFUSE RQ# Item_Name REJECT	Seller → Server	Seller refuses price reduction
	PRICE_ADJUSTMENT	PRICE_ADJUSTMENT RQ# Item_Name New_Price Time_Left	Server → Buyers	Broadcast adjusted price to all interested buyers
Auction Closure	WINNER	WINNER RQ# Item_Name Final_Price Seller_Name	Server → Winning Buyer	Buyer wins the auction
	SOLD	SOLD RQ# Item_Name Final_Price Buyer_Name	Server → Seller	Notify seller of successful sale
	NON_OFFER	NON_OFFER RQ# Item_Name	Server → Seller	Item did not receive any bid
Finalization	INFORM_Req	INFORM_Req RQ# Item_Name Final_Price	Server → Buyer & Seller	Request buyer/seller information for purchase
	INFORM_Res	INFORM_Res RQ# Name CC#	Buyer/Seller → Server	Submit transaction info

Category	Message Name	Message Format	Direction	Description
		CC_Exp_Date Address		
	CANCEL	CANCEL RQ# Reason	Server → Buyer & Seller	Transaction cancelled
	Shipping_Info	Shipping_Info RQ# Name Winner_Address	Server → Seller	Provide buyer's shipping details to seller

Table 1: System Design

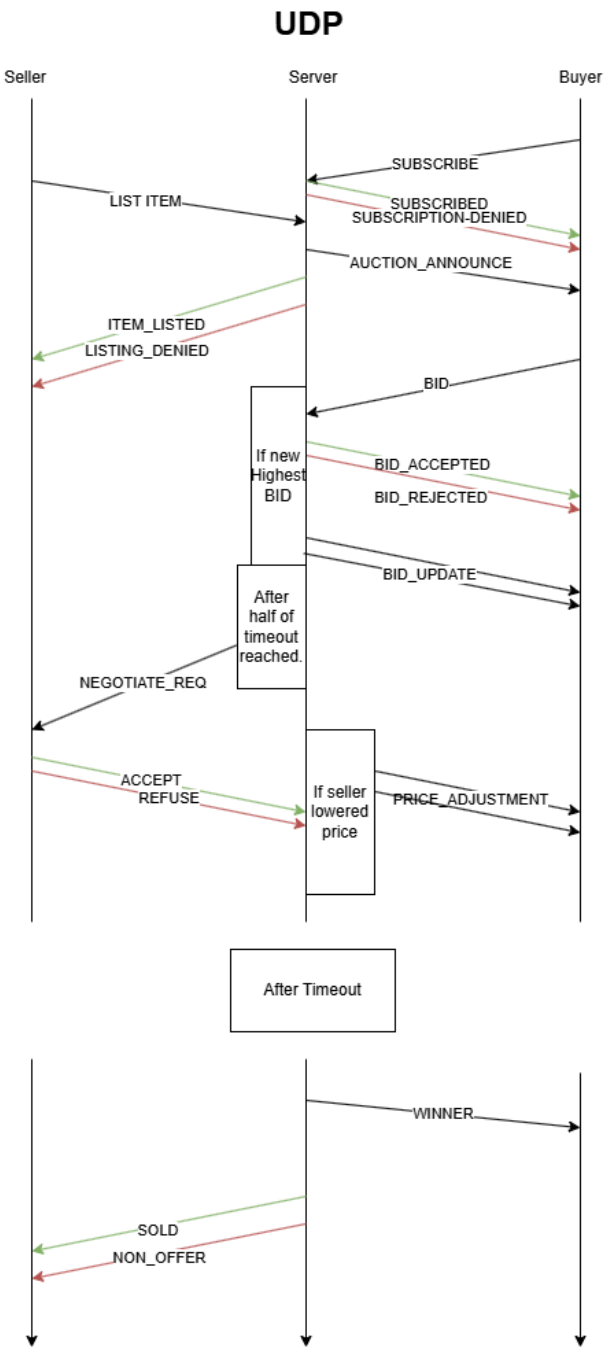


Figure 1: UDP System Design diagram

When critical data transmission and reliability are required—particularly at auction closure—the system switches from UDP to TCP. TCP ensures guaranteed delivery of important transactional messages such as winner notifications, payment processing, and shipping instructions. The server

initiates TCP connections to both buyers and sellers at the conclusion of an auction, coordinating transactions through a structured sequence of TCP messages to finalize payment details securely and efficiently.

4. Implementation

To implement the design, the client and server were separated into two distinct classes:

1. Auction Server
2. Auction Client

4.1 Server Implementation

The server class defined the ip address with a port number, a random UDP port, a random TCP port, and threads to listen for incoming messages. To handle the listeners, the “threading” native library was implemented to concurrently handle multiple client requests and responses at once. Python’s socket library was also utilized, it provides a low-level networking interface that allowed us to quickly set up the server hosting.

To handle different types of messages, the listener thread would read the message name and select the appropriate method using a switch statement. If the message was not recognized by the server, it would respond to the client with a DENIED message, with the reason stating it does not recognize the message name that was provided.

When a UDP message is received, it checks the validity of the message, which includes the number of parts in the message, as well as the content of the message. If it receives a valid message, it will process the content of the message depending on the method that was called. After the message was processed, if applicable the server would send a response to the client using the UDP port in their message, which would be an ACCEPTED or DENIED message with a request number and reason. When an item is created, a new thread is created to monitor the auction timer, and will perform an action when the timer has ended.

When a TCP message must be sent to the clients when finalizing an auction, the server will establish a TCP connection with a client using their defined TCP port they used when they registered with the service. Once a connection is established, they will reliably send the message to the client.

To provide a visual aid for the server actions, the server will log important events and errors that it encounters. Some of the logging events:

- Server running
- Thread creation & termination
- Message received
- Responses
- Errors

4.2 Client Implementation

When a new client is initialized, they will be prompted with a menu to navigate through the system. The client will be prompted to either login or register with the system, which will send the required information, including the UDP and TCP ports that they can be reached at alongside their IP address + port number.

When a client is logged in to the system, the client will create a new thread to listen for TCP messages. Therefore if a server wants to establish a TCP connection with them for auction results, there will be a thread awaiting a connection.

When a client sends a message, the socket that is awaiting a response from the server will set a timeout length of 5 seconds to handle any network delay. This ensures if the server sends a response that the client will not timeout before the response is received.

Similar to the server implementation, a logger has been implemented to provide a visual aid that the messages have been received and provide feedback for the user. Some of the logging events:

- Messages sent
- Responses

- Invalid formats
- Errors

5. Testing & Debugging

To thoroughly test the system, each message and different edge cases were tested. For each feature, we tested the main functionality and then tested the most common cases that would cause an error. For example, while listing an item we would test invalid inputs (such as writing a string into the duration input) to give us a LIST-DENIED message. This methodology was repeated throughout and applied to every test case.

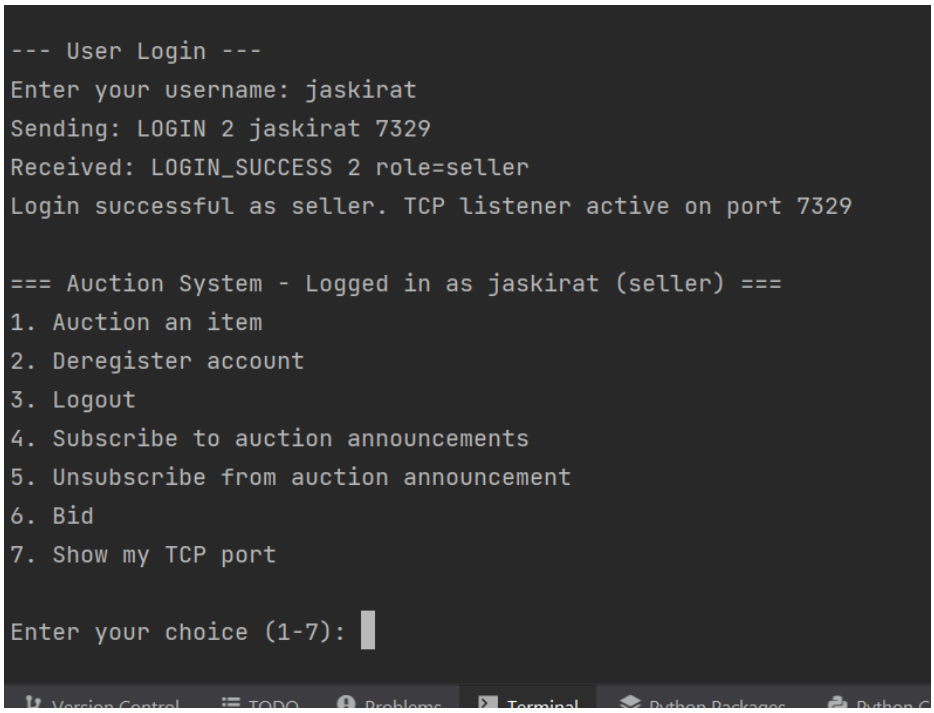
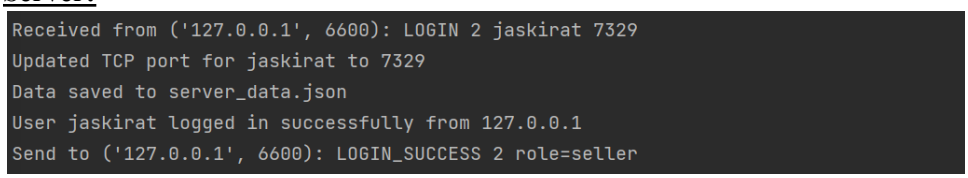
We individually tested every feature with a variety of test cases to ensure that the features were functioning properly. During this testing, we discovered a variety of different bugs. During the development of the TCP communications, we found out through our testing that on some features the TCP connection would attempt to connect to the wrong port due to a mistake in the creation of our threads. Our testing allowed us to easily find this bug and correct it.

The following section highlights the results of all the test cases:

5.1. Registration and Deregistration

Step	Log
REGISTER	**Client:** Send valid REGISTER message `=== Auction System ===` `1. Register` `2. Login` `3. Exit` `Enter your choice (1-3): 1` `Select your role (buyer/seller): seller` `TCP listener already running.` `Sending: REGISTER 1 jaskirat seller 192.168.2.24 6866 7845` `Received: REGISTERED 1` `Registration successful. TCP listener is active on port 7845` **Server:** Added user to JSON file and returned REGISTERED Response `Received from ('127.0.0.1', 6866): REGISTER 1 jaskirat seller 192.168.2.24 6866 7845` `Send to ('127.0.0.1', 6866): REGISTERED 1`
REGISTER-DENIED	**Server:** `Received from ('127.0.0.1', 6600): REGISTER 1 jaskirat seller 192.168.2.24 6600 7329` `Send to ('127.0.0.1', 6600): REGISTER-DENIED 1 User name is already taken` **Client:** `=== Auction System ===` `1. Register` `2. Login` `3. Exit` `Enter your choice (1-3): 1` `--- User Registration ---` `Enter your username: jaskirat` `Select your role (buyer/seller): seller` `Sending: REGISTER 1 jaskirat seller 192.168.2.24 6600 7329` `Received: REGISTER-DENIED 1 User name is already taken`

DE-REGISTER	**Server:** Send DE-REGISTER message `Received from ('127.0.0.1', 6600): DE-REGISTER 3 jaskirat` `Data saved to server_data.json` `User jaskirat deregistered` **Client:** No response `=== Auction System ===` `1. Register` `2. Login` `3. Exit` `2. Deregister account` `3. Logout` `4. Subscribe to auction announcements` `5. Unsubscribe from auction announcement` `6. Bid` `7. Show my TCP port` `Enter your choice (1-7): 2` `Are you sure you want to deregister? (y/n): y` `Sending: DE-REGISTER 3 jaskirat` `Deregistration message sent`

LOGIN	**Client:**  **Server:** 

<table>
<tr>
<td>LOGIN_FAIL</td>
<td>

```
Received from ('127.0.0.1', 6521): LOGIN 1 JASKIRAT] 7960
Login failed for user JASKIRAT] - not found
Send to ('127.0.0.1', 6521): LOGIN-FAILED 1 User not found
```

```
Enter your choice (1-3): 2

--- User Login ---
Enter your username: JASKIRAT]
Sending: LOGIN 1 JASKIRAT] 7960
Received: LOGIN-FAILED 1 User not found
Login failed. User not found or invalid credentials.
```

</td>
</tr>
<tr>
<td>BID

BID_ACCEPTED

BID_REJECTED</td>
<td>

Client:
Sends BID message with available item and valid bid

```
--- Bid on Item ---
Enter item name to bid on: games
Enter bid amount: 56
Sending Bid: BID 5 games 56
Received: BID_ACCEPTED 5
Bid of $56 accepted for games
```

Server:
Send BID_ACCEPTED response

```
Received from ('127.0.0.1', 6536): BID 5 games 56
Accepted bid of 56.0 from zack on games
Data saved to server_data.json
Send to ('127.0.0.1', 6536): BID_ACCEPTED 5
```

Client:
Sends BID message with unavailable item or invalid bid

</td>
</tr>
</table>

	`` --- Bid on Item --- Enter item name to bid on: food Enter bid amount: 3 Sending Bid: BID 5 food 3 Received: BID_REJECTED 5 Bid_too_low Bid rejected: Bid_too_low ``` **Server:** Send BID_REJECTED response with the attached reason ``` Received from ('127.0.0.1', 6521): BID 5 food 3 Send to ('127.0.0.1', 6521): BID_REJECTED 5 Bid_too_low ```
LIST_ITEM ITEM_LISTED LIST_DENIED	**Client:** Sends LIST_ITEM message with valid item details ``` --- Create Auction --- Enter item name: games Enter item description: to play Enter reserve price: 50 Enter auction duration in minutes: 2 Sending: LIST_ITEM 5 games to_play 50 2 jaskirat Received from server: ITEM_LISTED 5 ``` **Server:** Send ITEM_LISTED response

```
Received from ('127.0.0.1', 6754): LIST_ITEM 2 food hungry 34 2 jaskirat
Data saved to server_data.json
Auction for food will end in 120.00 seconds
Send to ('127.0.0.1', 6754): ITEM_LISTED 2
```

Client:
Sends LIST_ITEM message with invalid item details

```
--- Create Auction ---
Enter item name: food
Enter item description: food
Enter reserve price: 4
Enter auction duration in minutes: 2
Sending: LIST_ITEM 2 food food 4 2 jame
Received: LIST_DENIED 2 item name already exists
```

Server:
Send LIST_DENIED response with the attached reason

```
Received from ('127.0.0.1', 6734): LIST_ITEM 2 food food 4 2 jame
Send to ('127.0.0.1', 6734): LIST_DENIED 2 item name already exists
```

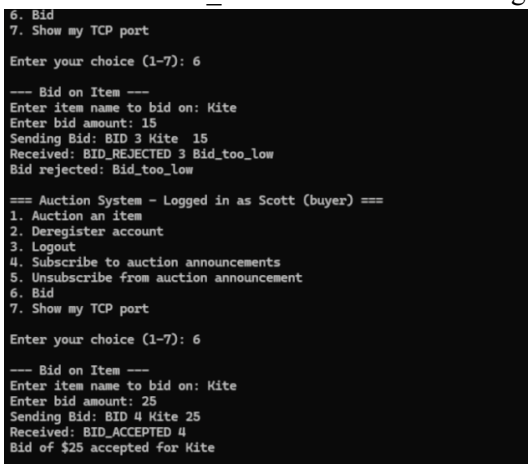
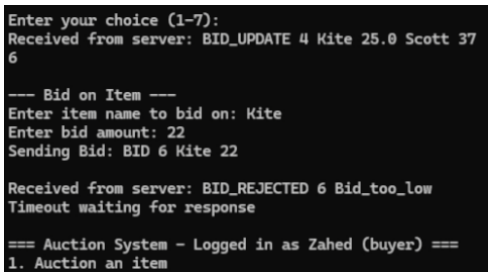
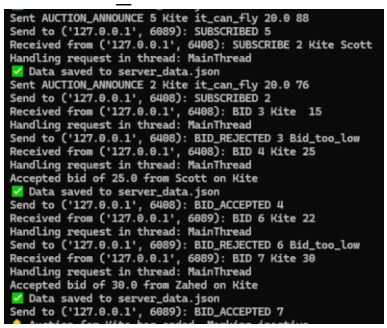
AUCTION_ANNOUNCEMNT	**Client:** SUBSCRIBE message sent with valid item name `=== Auction System - Logged in as zack (buyer) ===` `1. Auction an item` `2. Deregister account` `3. Logout` `4. Subscribe to auction announcements` `5. Unsubscribe from auction announcement` `6. Bid` `7. Show my TCP port` `Enter your choice (1-7): 4` `--- Subscription ---` `Enter item name: food` `Sending: SUBSCRIBE 6 food zack` `Received from server: SUBSCRIBED 6` **Server:** Send SUBSCRIBED response `Received from ('127.0.0.1', 6521): SUBSCRIBE 6 food zack` `Data saved to server_data.json` `Send to ('127.0.0.1', 6521): SUBSCRIBED 6` **Client:** Sends SUBSCRIBE message with invalid item name

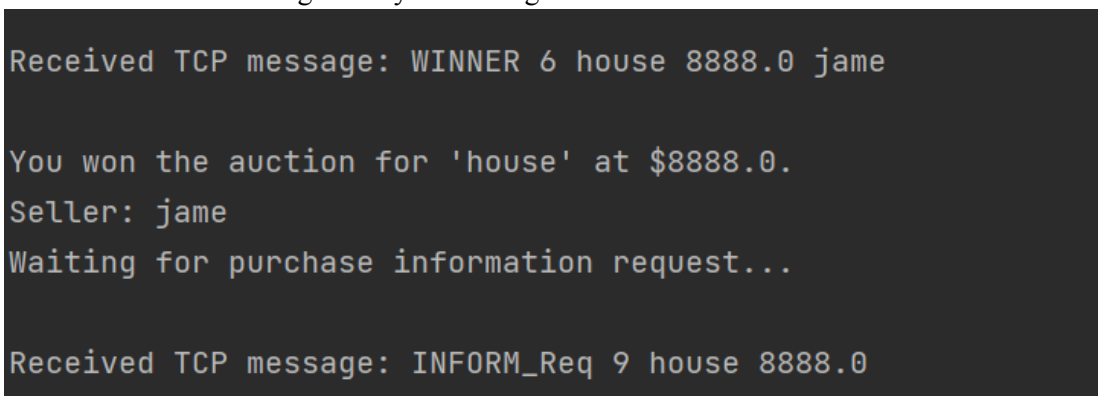
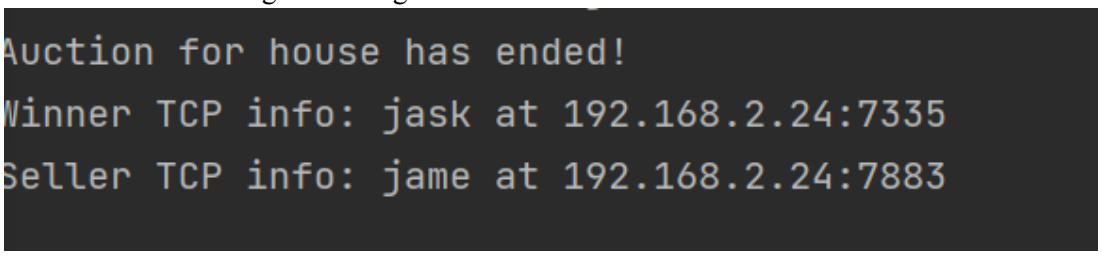
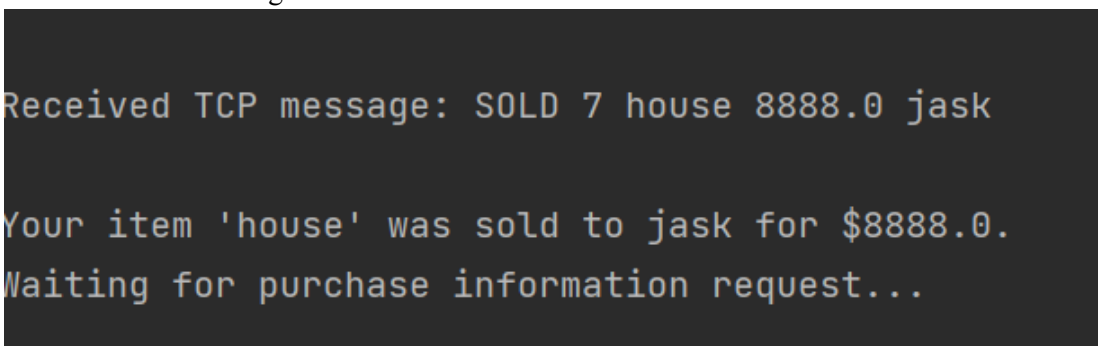
```
=== Auction System - Logged in as zack (buyer) ===
1. Auction an item
2. Deregister account
3. Logout
4. Subscribe to auction announcements
5. Unsubscribe from auction announcement
6. Bid
7. Show my TCP port

Enter your choice (1-7): 4

--- Subscription ---
Enter item name: pen
Sending: SUBSCRIBE 8 pen zack
Received: SUBSCRIPTION-DENIED 8 item does not exist
Subscription denied: item does not exist
```

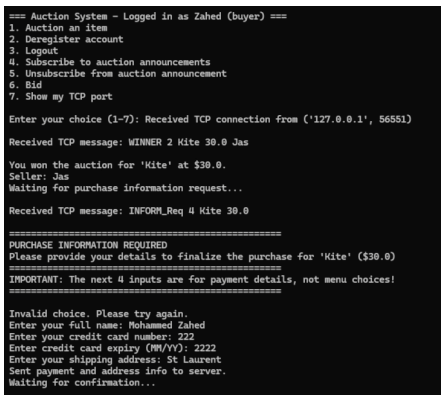
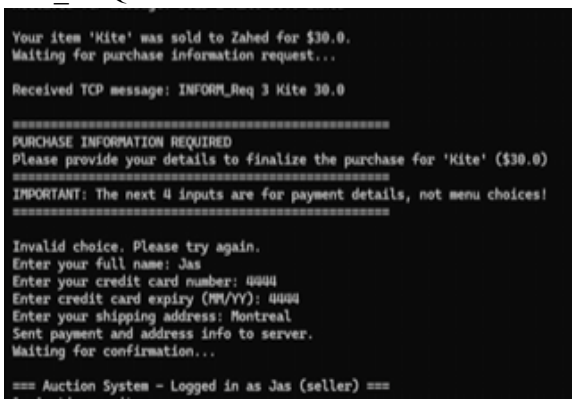
Server:

Send SUBSCRIPTION-DENIED response with the attached reason

```
Received from ('127.0.0.1', 6521): SUBSCRIBE 8 pen zack
Send to ('127.0.0.1', 6521): SUBSCRIPTION-DENIED 8 item does not exist
```

BID_UPDATE	**Client:** Receives the BID_UPDATE when a new highest bid is made on a subscribed item   **Server:** Send BID_UPDATE to all clients that are subscribed to the item 

WINNER SOLD NON_OFFER	**Client:** Receive WINNER message if they are the highest bid for an item  **Server:** Send WINNER message to the highest bidder once the auction ends  **Client:** Receives SOLD message from the server  **Server:** Sends SOLD message to client who is the seller of the item when the auction ends

	Sent to buyer jask: WINNER 6 house 8888.0 jame Sent to seller jame: SOLD 7 house 8888.0 jask Sending INFORM_Req to seller jame Sent to seller jame: INFORM_Req 8 house 8888.0 Sending INFORM_Req to buyer jask Sent to buyer jask: INFORM_Req 9 house 8888.0 NON_OFFER Data saved to server_data.json Auction for food has ended! NO BIDS !!! Sent to seller jaskirat: NON_OFFER 5 food
INFORM_REQ INFORM_RES CANCEL SHIPPING_INFO	**Server:** TCP connection established and INFORM_REQ sent   **Client:** Accept TCP request and sends message INFORM_RES  **Server:** TCP connection established and SHIPPING_INFO sent **Client:**

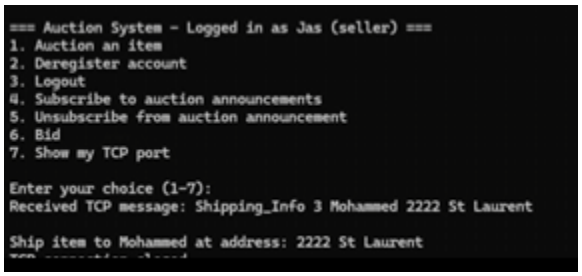
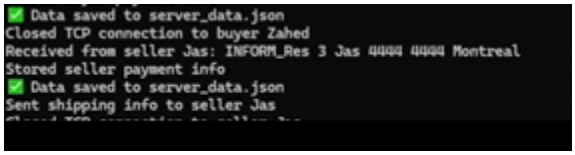
	Accept TCP request and sends message SHIPPING_INFO  

Table 2: Test cases

6. Conclusion

During this project, our team developed a much deeper understanding of how network programming is achieved and how communication protocols work. We learned in detail how threads work and how they are used in a networking context as well as the differences between UDP and TCP in a practical environment.

One thing the team had trouble with during the implementation was the TCP connection. Establishing a connection with the client proved to be difficult, since we discovered that the client always had to have a thread active listening for incoming TCP requests.

All of the team members worked on the server and thread setup, with individuals contributing the following:
Scott: Registration/deregistration, server setup, client menus + setup, auction subscription
Zahed: Auction announcements subscription, server setup, threads setup, auction/bidding (TCP + UDP)

Jaskirat: Auction closure and finalization (TCP), server setup, threads setup, auction updates

Overall this project was a success for our group, and a lot was learned throughout the process.

APPENDIX

The demo is provided here and within the submission.

2025-04-12 22-49-41.mp4