# Exp 0

I need two complete Python programs to support a client-server model using Linux sockets for a ticketing system. The programs will develop a two-tiered ticketing system that will consist of a ticket outlet (i.e., the server) that will provide ticket services such as “BUY” and “SELL” tickets to two local ticket distributors (i.e., the clients), who will also serve as “scalpers”, providing tickets at double the advertised cost to the “buyer”, which is the other client.

## Exp 0-1

* The server will:

• generate N tickets, where N = 25, and each ticket will have a unique 5-digit number. Each ticket will cost between $200 and $400, inclusively, using a seeded, randomly generated integer to assign the value and initially be made available for sale. In this case, the server will therefore maintain a database of 25 tickets that contains the ticket number, the ticket value, and the availability of the ticket (i.e., whether or not it has been sold or is available). The entire database will be printed before any transactions (where all tickets are available), every time a transaction took place, and after the last transaction has been completed and both clients have been disconnected.

• Set up an Internet domain TCP socket using the port passed into the server program and listen to the socket for both clients to communicate on that socket. The server must wait for both clients to connect to the socket before processing any (read) requests. Once both clients have connected, however, the server should respond equitably as each request simultaneously comes from different clients.

• Support two types of requests:

1. BUY <user balance>

When a client sends a BUY request with the user’s monetary balance, the server can respond in one of three ways:

<ticket number> <ticket price>

If the server has tickets available for sale and the user has sufficient funds in his/her user account, then the server will respond with the unique 5-digit ticket number being purchased along with the price of the ticket. After the ticket has been sold to the client, the server will update the database to indicate that the ticket has been sold.

1. NOFUNDS

If the server has tickets available for sale, but the user does not have sufficient funds in his/her user account, then the server will respond with a NOFUNDS message. Keep in mind that since users can SELL or SCALP tickets, the user may acquire funds to allow a future BUY request to be successful.

1. SOLDOUT

If the server has no tickets available (i.e., they are currently all sold), then the server will respond with a SOLDOUT message. Keep in mind that since the server supports a SELL request, tickets may alternate between being sold out and not.

1. SELL <ticket number>

When a client sends a SELL request with the ticket number, then the server will respond with a response indicating that the SELL request was successful (i.e., <ticket number> <ticket original price>). After the ticket has been sold to the server, the server will update the database to indicate that the ticket is now available for sale.

* After both clients have disconnected, the server will print the current state of the database and close the sockets.

ChatGPT response:

To design the server portion of the ticketing system using Linux sockets, we need to follow several steps to ensure that it meets the specifications provided. Here is an outline of the steps required to implement the server functionality:

**1. Initialize the Server**

* **Set up the socket:** The server will create an Internet domain TCP socket and bind it to a port provided as an argument to the server program.
* **Listen for connections:** Set the socket to listen mode and prepare to accept connections from clients.

**2. Generate and Manage Tickets**

* **Generate tickets:** Create a list of 25 tickets where each ticket has a unique 5-digit number and a random ticket price between $200 and $400. This can be achieved by using a random number generator with a fixed seed for reproducibility.
* **Ticket database:** Maintain a list (or dictionary) to store ticket details including the ticket number, price, and availability status (sold or available).

**3. Handling Client Connections**

* **Wait for clients:** The server should wait until both clients are connected before processing any requests.
* **Manage connections:** Store client information and handle each connection in a separate thread to allow simultaneous processing of requests.

**4. Process Client Requests**

* **Request types:** Handle different types of requests from clients:
  + **BUY:** Check if the ticket is available and if the client has enough funds. Respond with the ticket number and price, mark the ticket as sold, or send appropriate error messages (NOFUNDS or SOLDOUT).
  + **SELL:** Accept the ticket number to be sold back to the server, update its status to available, and confirm the transaction with the client by sending back the ticket number and its original price.

**5. Maintain a Transaction Log**

* **Transaction log:** Keep a record of all transactions, including buys and sells, and update the ticket database accordingly.
* **Print database state:** Print the current state of the ticket database initially, after each transaction, and after the last transaction when both clients disconnect.

**6. Clean Up and Close**

* **Close client connections:** Properly close each client connection after their transactions are complete.
* **Shutdown server:** After both clients have disconnected, print the final state of the database and close the server socket.

**7. Error Handling and Validation**

* **Error handling:** Implement error handling for situations like client disconnection, invalid requests, or socket errors.
* **Validation:** Ensure that the data received from clients is validated before processing (e.g., checking if a BUY or SELL request contains valid and expected data).

**8. Multithreading and Synchronization**

* **Thread safety:** Since multiple clients might access and modify the ticket database concurrently, ensure that these operations are thread-safe using locks or other synchronization mechanisms.

These steps outline the basic structure and functionality required for the server to handle a ticketing system efficiently. Implementing these carefully will ensure that the server is robust and can manage client requests effectively in a multi-client environment.

## 2. Exp 0-2:

* There are 2 clients, but only 1 client source code. The clients will:
* Initially be given a user balance of $4,000 to be used in BUY, SELL, and SCALP requests.
* The client will maintain a database of 15 tickets that contains the ticket number and the ticket value. The entire database plus the remaining user balance will be printed after the last transaction has been completed from both clients (hint: you may want to sleep a couple of seconds to allow the other client’s transactions to complete as well).
* The client will connect to the Internet domain TCP socket set up by the server using the hostname and port passed into the client program. This socket connection will be used to issue 15 automated BUY and/or SELL requests to the server (details on how to proceed are described below).
* The client will then set up an Internet domain UDP socket using the IP address and value of one greater than the port number (i.e., port number + 1) passed in to the client program and listen to the socket for the other active client to communicate on that socket. Note that since the client must simultaneously accomplish to different tasks (i.e., sending automated requests to the server AND listening to the Internet domain UDP socket), you are encouraged to either create a child process or thread to support this functionality. This Internet domain UDP socket will allow the client to behave as a server to the other client, functioning as a “scalper”, accepting SCALP requests from a “buyer”
* The client will send 15 automated BUY requests to the server in a format: BUY <user balance>. The client will behave based on the server’s respond to each BUY request. If the server’s respond is:
  1. <ticket number> <ticket price>  
     The client will update the client’s database with the newly acquired ticket information and deduct the purchase price from his/her user balance.
  2. NOFUNDS  
     The client will then select any ticket he/she chooses and send a SELL request back to the server to sell the ticket back to the server at the original purchase price to acquire additional funds. After the ticket has been sold to the server, the client will remove the corresponding entry from its database and update the client’s user balance with the money received from the SELL request.
  3. SOLDOUT  
     The client will then transform its behavior to a “buyer”, which means that he/she will then conduct a transaction with a “scalper”, which is essentially the “other” client. The client will send a SCALP request to the “scalper” (i.e., the other client).
* The client can send a SCALP request to the other client in the format of: SCALP <user balance>.  
  The client will need to act based on the scalper’s response, which can be one of two ways:

1. <ticket number> <two times the ticket price>  
   If the “scalper” responds with the message, then the “buyer” will update the client’s database with the newly acquired ticket information and deduct the purchase price from his/her user balance.
2. NOMONEY  
   If the “scalper” responds with a NOMONEY message, the “buyer” will then select any ticket he/she chooses and send a SELL request back to the server to sell the ticket back to the server at the original purchase price to acquire additional funds. After the ticket has been sold to the server, the client will remove the corresponding entry from its database and update the client’s user balance with the money received from the SELL request.

* After the client has completed its 15 automated BUY requests, the client will print the current state of the database along with the remaining user balance and close the socket.

Don’t implement the code yet. We need to outline the steps that must be done for the client to function as required.

ChatGPT response:

To implement the client program for the ticketing system as described, we need to outline several key steps to ensure the client can properly interact with the server and another client for the ticket transactions. Here is a structured outline for the client's operations:

**1. Initialize the Client**

* **Set up initial variables:** Initialize a user balance of $4,000 and a local ticket database capable of storing up to 15 tickets.
* **Connect to the server:** Use the provided hostname and port to establish a TCP connection with the server.

**2. Handle Concurrent Operations**

* **Set up threading or multiprocessing:** Since the client needs to handle automated requests to the server and listen to another client simultaneously, use a separate thread or process. This allows the client to manage TCP requests/responses with the server while also operating a UDP server to handle incoming SCALP requests.
* **UDP socket setup:** Establish a UDP socket on the port number given to the client program +1. This will be used to listen to SCALP requests from the other client.

**3. Automated Buy/Sell Requests to Server**

* **Send BUY requests:** For each of the 15 transactions, send a BUY request to the server with the current user balance. Depending on the server's response, the client will:
  + **Ticket and price received:** Update the local database with the new ticket, adjust the user balance, and continue to the next request.
  + **NOFUNDS:** Choose a ticket to sell, send a SELL request to the server, update the user balance with the received funds, and remove the ticket from the local database.
  + **SOLDOUT:** Transition to buying tickets from the other client (scalper) using SCALP requests.

**4. Scalping Functionality**

* **Handle incoming SCALP requests:** As a server (via UDP), receive SCALP requests, decide on the availability and price of tickets (double the initial price), and respond appropriately.
* **Act as a buyer:** If required, send SCALP requests to the other client, handle responses, update the local database and user balance accordingly.

**5. Conclude Transactions**

* **Print final status:** After completing all transactions, print the current state of the ticket database and the remaining user balance.
* **Close sockets:** Properly close both the TCP and UDP sockets to ensure there are no resource leaks.

**6. Error Handling and Synchronization**

* **Manage network errors:** Handle potential network issues like disconnections or timeouts gracefully.
* **Synchronize access:** If using threads, ensure that access to shared resources like the user balance and ticket database is synchronized to prevent data corruption.

**7. Cleanup and Exit**

* **Ensure clean exits:** Make sure that the client can gracefully handle termination signals (like SIGINT) to exit cleanly without leaving orphaned processes or open sockets.

These steps provide a comprehensive roadmap for implementing the client part of your ticketing system, ensuring that it can handle automated transactions with the server and peer-to-peer interactions effectively.

## Exp 0-3:

Ask ChatGPT to implement steps 1 to 3 in server outline.

Prompt for logging output: Please incorporate logging in the program. Make sure to log every output that is printed out on the screen, any errors, any exceptions, any messages received from clients, and messages sent out by the server to the clients. Save the log in the format of <program\_name.log>

Strategy: Provide the requirements of the server and client programs separately, and ask ChatGPT to outline the steps to do before the actual implementation.

Currently: ask ChatGPT to implement the server program and complete its functionalities before moving on to implementing client program.

## Exp 0-4:

Ask ChatGPT to implement steps 4-6 based on the server outline

## Exp 0-5:

Ask ChatGPT to include logging function to record the output

## Exp 0-6:

Prompt for client-0-6.py:

Now we will focus on the client program. Based on the outline that you generated for the client program. Please implement steps 1 to 3 in Python to create the base for the client:  
1. Initialize the Client

Set up initial variables: Initialize a user balance of $4,000 and a local ticket database capable of storing up to 15 tickets.

Connect to the server: Use the provided hostname and port to establish a TCP connection with the server.

2. Handle Concurrent Operations

Set up threading or multiprocessing: Since the client needs to handle automated requests to the server and listen to another client simultaneously, use a separate thread or process. This allows the client to manage TCP requests/responses with the server while also operating a UDP server to handle incoming SCALP requests.

UDP socket setup: Establish a UDP socket on the port number given to the client program +1. This will be used to listen to SCALP requests from the other client.

3. Automated Buy/Sell Requests to Server

**Send BUY requests:** For each of the 15 transactions, send a BUY request to the server with the current user balance. Depending on the server's response, the client will:

* **Ticket and price received:** Update the local database with the new ticket, adjust the user balance, and continue to the next request.
* **NOFUNDS:** Choose a ticket to sell, send a SELL request to the server, update the user balance with the received funds, and remove the ticket from the local database.
* **SOLDOUT:** Transition to buying tickets from the other client (scalper) using SCALP requests.

Errors: there are errors occurred in both sides.

1) Server did not wait for the second client to connect, but proceeded to selling tickets to the first client.

2) Client 1 finished first and exited, server exited, and Client 2 was still idling indefinitely

3) In a message containing the ticket ID and price, the server sent: "1000 3411000 341", causing an error of too many values to unpack.

## Exp 0-7:

Fix error 1: Server did not wait for the second client to connect, but proceeded to selling tickets to the first client.

Prompt: Attached is the server program that you implemented. However, there are several errors. The first error was that the server did not wait for the second client to connect, but proceeded to sell tickets to the first client.   
The server is supposed to wait for both clients to connect and then start processing the sell requests from both clients. Please fix this problem given the server program that you implemented.

## Exp 0-8:

Goal: Fix error 2: Client 1 finished and exited, the server then exited, and client 2 was still idling indefinitely.

Prompt: Attached are the server program and the client program that you implemented. However, there are still errors. After completing the transactions, the first client exited, and the server did not wait for the second client to exit, but proceeded to close down the connection, leaving the second client idling indefinitely.

The server is supposed to wait for both clients to exit, then log the final ticket database, and close down the connection and exit only when the other client exits. Please fix this problem given the server program and the client program that you implemented.

Please be sure to log all messages that were sent or received by both server and client. Also, the log mode should be DEBUG to record all levels of information during execution.

Errors: there are errors occurred in both sides.

1) Server did not wait for the second client to connect, but proceeded to selling tickets to the first client. 🡪 error 1 repeats

2) Client 1 finished first and exited, the server exited, and Client 2 was still idling indefinitely 🡪 not fixed

3) In a message containing the ticket ID and price, the server sent: "1000 3411000 341", causing an error of too many values to unpack. -🡪 does not happen any more.

4) Error in UDP connection: client-0-6.py", line 29, in udp\_listener udp\_sock.bind((hostname, udp\_port)) OSError: [WinError 10048] Only one usage of each socket address (protocol/network address/port) is normally permitted

## Exp 0-9:

Goal: Fix error 1 and 2 at the same time.

Prompt: The revised server program that you just generated performed worse than before. The server is supposed to wait for both clients to enter before beginning to sell tickets. Also, the server is supposed to wait for both clients to exit before closing down the connections and exiting the program. Attached is the server program that you generated before last, in which, the server was waiting for both clients to join before processing their buy requests. Based on this server program, address the second issue, which is waiting for both clients to exit before closing down the connections and exiting the program.

Please be sure to log all messages that were sent and received by both of the server and clients. Also, the log mode should be DEBUG to record all levels of information during execution.