



# DISTRIBUTED COMPUTING EXPERIMENT 1

**Submitted By:**

**Name: Adwait Purao**

**UID:** 2021300101

**Batch:** B2

# Submitted To:

Prof. Pramod Bide

## Aim:

Implementation of Client-Server Communication using Sockets

## Theory:

### Sockets Overview:

Sockets are a fundamental communication mechanism that enables data exchange between processes or applications running on different devices over a network. In client-server architecture, sockets are used to establish connections and facilitate communication.

### Components:

**Client:** The client is the application or system that initiates a connection request to the server. It typically requests services or data from the server.

**Server:** The server is responsible for listening to incoming connection requests from clients and providing services or data as requested.

### Socket Types:

There are two primary socket types used in client-server communication:

1. **TCP Sockets:** Transmission Control Protocol (TCP) sockets provide reliable, connection-oriented communication. They ensure data integrity and order but may have higher overhead.
2. **UDP Sockets:** User Datagram Protocol (UDP) sockets offer faster, connectionless communication but do not guarantee data integrity or order. They are suitable for scenarios where speed is prioritized over reliability.

### Key Steps in Client-Server Communication:

**Server Initialization:** The server creates a socket, binds it to a specific IP address and port, and starts listening for incoming client connections.

**Client Connection:** The client creates a socket and initiates a connection to the server's IP address and port.

### Data Exchange:

1. **Server Reception:** The server accepts incoming client connections, creating a new socket for communication. It receives data from the client through this socket.
2. **Client Transmission:** The client sends data to the server using its socket.
3. **Data Processing**: The server processes the received data, performs requested operations (e.g., handling inventory requests), and may send a response back to the client.
4. **Connection Termination:** Both client and server can close their sockets when the communication is complete.
5. **Error Handling:** Effective error handling is crucial in socket communication. It involves dealing with issues such as connection failures, data transmission errors, and unexpected disconnections.
6. **Scalability and Performance:** The experiment may explore how well the client-server architecture scales under various loads, considering factors like concurrency, response times, and resource utilization.
7. **Security:** Security measures like encryption and authentication may be implemented to protect data privacy and system integrity.
8. **Socket Libraries:** Depending on the programming language, libraries like Python's socket, Java's Socket and Server Socket, or Node.js's net module can be used to implement sockets.

## Code:

**Server.py:**

import socket import pickle

# Sample inventory data (in-memory storage) inventory = {}

def add\_item(item\_name, quantity): if item\_name in inventory:

inventory[item\_name] += quantity else:

inventory[item\_name] = quantity

def view\_inventory(): return inventory

def main():

server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) server.bind(('127.0.0.1', 12345))

server.listen(5)

print("Server is listening...")

while True:

client\_socket, client\_address = server.accept() print(f"Accepted connection from {client\_address}")

request = client\_socket.recv(1024) request = pickle.loads(request)

if request['action'] == 'add': item\_name = request['item\_name'] quantity = request['quantity'] add\_item(item\_name, quantity)

response = {'message': f"Added {quantity} {item\_name}(s) to the inventory."}

elif request['action'] == 'view':

response = {'inventory': view\_inventory()}

client\_socket.send(pickle.dumps(response)) client\_socket.close()

if name == " main ": main()

Client.py:

import socket import pickle

def send\_request(request):

client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) client.connect(('127.0.0.1', 12345))

client.send(pickle.dumps(request)) response = client.recv(1024)

response = pickle.loads(response) print(response)

client.close()

def main(): while True:

print("\nOptions:")

print("1. Add Item to Inventory") print("2. View Inventory") print("3. Exit")

choice = input("Enter your choice: ")

if choice == '1':

item\_name = input("Enter item name: ")

quantity = int(input("Enter quantity to add: "))

request = {'action': 'add', 'item\_name': item\_name, 'quantity':

quantity}

send\_request(request) elif choice == '2':

request = {'action': 'view'} send\_request(request)

elif choice == '3': break

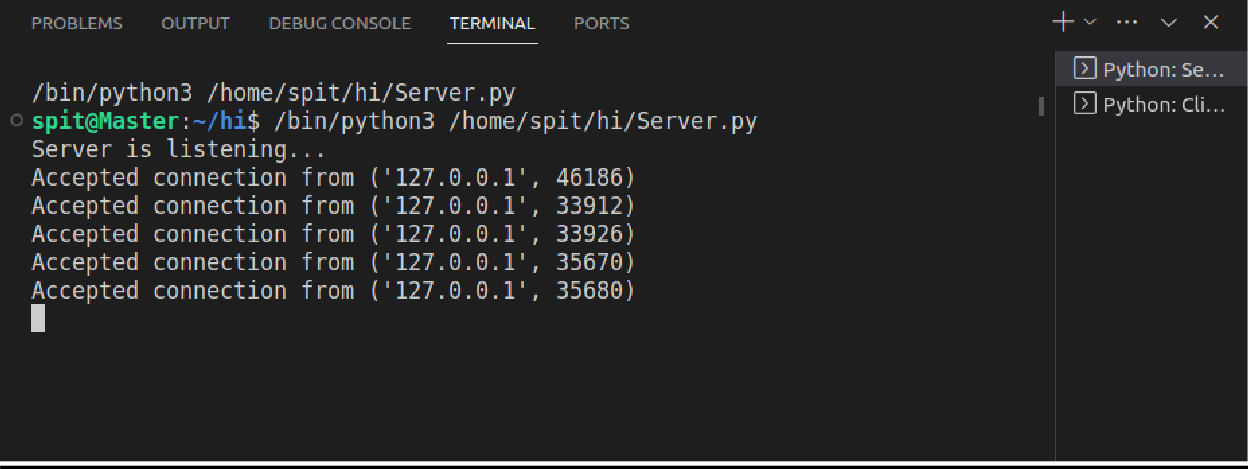
else:

print("Invalid choice. Try again.")

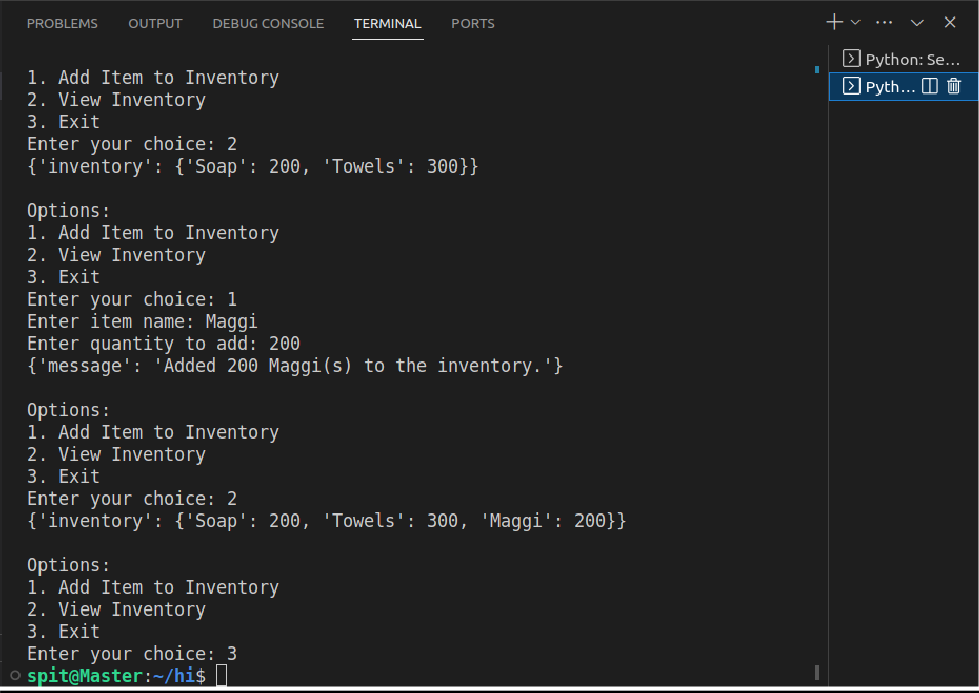
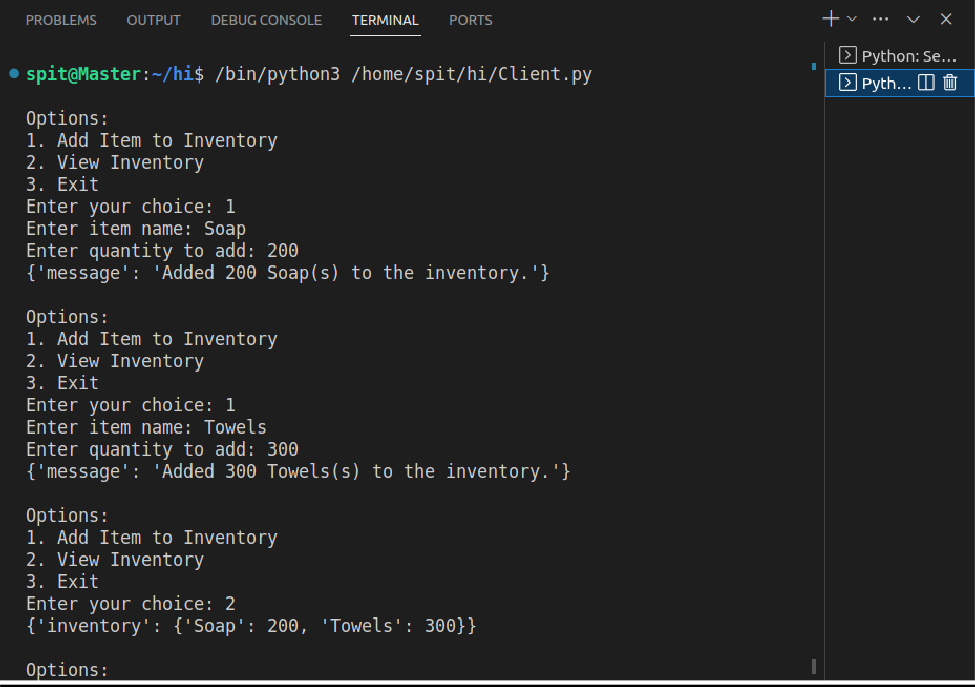
if name == " main ": main()

## Output:

Server:



Client:



## Conclusion:

In summary, this experiment centered on implementing client-server communication using sockets, a crucial networking mechanism. It covered the core components of client-server architecture, socket types (TCP and UDP), and the essential communication steps.

Implementing client-server communication using sockets is a fundamental technique for building networked applications.

The Python code demonstrated a practical application of socket programming with a server managing an inventory system.

Socket programming remains a fundamental skill for networking and distributed applications, highlighting the importance of efficient communication in modern computing environments.