

Adwait Purao – 2021300101 TE Comps B - Batch B

DC LAB

**DISTRIBUTED COMPUTING EXP 4**

**Aim:**

Implementation of multi-threading in distributed systems

**Theory:**

**Introduction:**

Distributed systems are composed of multiple interconnected computers that work together to achieve a common goal. They are often designed to handle large-scale applications and require efficient utilization of computing resources. Multi-threading is a programming technique that can enhance the performance and responsiveness of distributed systems by allowing concurrent execution of tasks within a single process or across multiple processes.

**Multi-Threading:**

Multi-threading is a mechanism that enables a single process to have multiple threads of execution. Each thread is a lightweight unit of execution that shares the same memory space, allowing them to communicate and coordinate with each other efficiently. In the context of distributed systems, multi-threading offers several advantages:

1. **Parallelism:** Multi-threading allows different threads to execute tasks simultaneously. This parallelism can significantly improve the overall system throughput, especially when dealing with computationally intensive or I/O-bound operations.
2. **Responsiveness:** In distributed systems, responsiveness is crucial. Multi-threading can keep a system responsive by allowing certain threads to handle user interactions, while others handle background tasks, such as network communication or data processing.
3. **Resource Utilization:** Multi-threading enables efficient utilization of CPU cores and memory resources. This is essential in distributed systems where resource optimization is a primary concern.
4. **Synchronization:** To maintain data integrity and consistency in a distributed environment, threads need to synchronize their activities. Proper synchronization mechanisms, such as locks, semaphores, or message passing, are essential to ensure that multiple threads do not interfere with each other's operations.

**Challenges in Multi-Threading in Distributed Systems:**

While multithreading offers numerous benefits, it also introduces challenges in the context of distributed systems:

1. **Concurrency Control:** Coordinating threads across multiple distributed nodes can be complex. Ensuring that threads do not access shared resources concurrently and maintaining consistency requires careful design and synchronization.
2. **Fault Tolerance:** Distributed systems are susceptible to failures, such as network outages or node crashes. Managing threads in a way that maintains system integrity and recovers gracefully from failures is a non-trivial task.
3. **Scalability:** As distributed systems grow in size, managing an increasing number of threads can become challenging. Ensuring scalability and efficient load balancing is vital.

**Best Practices for Multi-Threading in Distributed Systems:** To successfully implement multithreading in distributed systems, consider the following best practices:

1. **Thread Pooling:** Use thread pools to manage and control the number of threads. This helps prevent resource exhaustion and simplifies thread lifecycle management.
2. **Isolation:** Isolate critical sections of code and use appropriate synchronization mechanisms to prevent data corruption and race conditions.
3. **Error Handling:** Implement robust error handling and fault tolerance mechanisms to handle failures gracefully and recover from them.
4. **Load Balancing:** Implement load balancing algorithms to distribute tasks evenly across nodes and threads, ensuring efficient resource utilization.
5. **Monitoring and Profiling:** Use monitoring and profiling tools to identify performance bottlenecks and fine-tune the multi-threaded code for optimal performance.

Multi-threading is a powerful technique that can enhance the performance and responsiveness of distributed systems. However, it comes with challenges that require careful consideration and design. When implemented correctly, multi-threading can lead to more efficient and scalable distributed systems, ultimately delivering a better user experience and improved resource utilization.

**Code:**

**Server:**

from xmlrpc.server import SimpleXMLRPCServer, SimpleXMLRPCRequestHandler

from socketserver import ThreadingMixIn

import threading

# Inventory management class

class InventoryManager:

def \_\_init\_\_(self):

self.inventory = {}

self.product\_id\_counter = 1

def add\_item(self, item\_name, quantity):

product\_id = self.product\_id\_counter

self.inventory[product\_id] = {"name": item\_name, "quantity": quantity}

self.product\_id\_counter += 1

return f"Product added with ID {product\_id}\n"

def update\_item(self, product\_id, quantity):

if product\_id in self.inventory:

self.inventory[product\_id]["quantity"] = quantity

return f"Product with ID {product\_id} quantity updated successfully\n"

else:

return f"Product with ID {product\_id} not found\n"

def delete\_item(self, product\_id):

if product\_id in self.inventory:

del self.inventory[product\_id]

return f"Product with ID {product\_id} deleted successfully\n"

else:

return f"Product with ID {product\_id} not found\n"

def get\_item(self, product\_id):

if product\_id in self.inventory:

product = self.inventory[product\_id]

return f"Product ID: {product\_id}, Name: {product['name']}, Quantity: {product['quantity']}\n"

else:

return f"Product with ID {product\_id} not found"

def get\_all\_items(self):

# Convert product IDs to strings for compatibility with XML-RPC

str\_inventory = {str(key): value for key, value in self.inventory.items()}

return str\_inventory

# Custom request handler class with threading enabled

class ThreadedXMLRPCRequestHandler(SimpleXMLRPCRequestHandler):

rpc\_paths = ('/RPC2',)

# Threaded XML-RPC server class

class ThreadedXMLRPCServer(ThreadingMixIn, SimpleXMLRPCServer):

pass

# Function to handle client requests

def handle\_client(client, addr):

print(f"Accepted connection from {addr}")

try:

while True:

client.handle\_request()

except Exception as e:

print(f"Error: {e}")

finally:

print(f"Connection with {addr} closed.")

# Server configuration

server = ThreadedXMLRPCServer(("localhost", 9090), requestHandler=ThreadedXMLRPCRequestHandler)

server.register\_introspection\_functions()

# Create an instance of the InventoryManager class

inventory\_manager = InventoryManager()

# Register the InventoryManager methods for RPC

server.register\_function(inventory\_manager.add\_item, "add\_item")

server.register\_function(inventory\_manager.update\_item, "update\_item")

server.register\_function(inventory\_manager.delete\_item, "delete\_item")

server.register\_function(inventory\_manager.get\_item, "get\_item")

server.register\_function(inventory\_manager.get\_all\_items, "get\_all\_items")

print("Server is listening on port 9090...")

# Start the server in a separate thread

server\_thread = threading.Thread(target=server.serve\_forever)

server\_thread.start()

# Wait for the server thread to finish

server\_thread.join()

print("Server closed.")

**Client:**

import xmlrpc.client

import threading

# Function to interactively manage inventory using product IDs

def interactively\_manage\_inventory(proxy):

while True:

print("Choose an action:")

print("1. Add item")

print("2. Update item quantity")

print("3. Delete item")

print("4. Get item quantity")

print("5. Display all items")

print("6. Quit")

print("")

choice = input("Enter your choice: ")

if choice == "1":

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

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

result = proxy.add\_item(item\_name, quantity)

print(result)

elif choice == "2":

item\_id = int(input("Enter the product ID to update quantity: "))

quantity = int(input("Enter the new quantity: "))

result = proxy.update\_item(item\_id, quantity)

print(result)

elif choice == "3":

item\_id = int(input("Enter the product ID to delete: "))

result = proxy.delete\_item(item\_id)

print(result)

elif choice == "4":

item\_id = int(input("Enter the product ID to get quantity: "))

item\_info = proxy.get\_item(item\_id)

print(item\_info)

elif choice == "5":

all\_items = proxy.get\_all\_items()

if all\_items:

print("All Items in Inventory: ")

for product\_id, product\_info in all\_items.items():

print(f"Product ID: {product\_id}, Name: {product\_info['name']}, Quantity: {product\_info['quantity']}")

print("")

else:

print("Inventory is empty.")

elif choice == "6":

print("Thank You for using our Services\n")

break

else:

print("Invalid choice. Please try again.")

# Function to handle client communication in a separate thread

def client\_thread():

proxy = xmlrpc.client.ServerProxy("http://localhost:9090/RPC2", allow\_none=True)

interactively\_manage\_inventory(proxy)

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

# Start a separate thread for client communication

client\_thread = threading.Thread(target=client\_thread)

client\_thread.start()

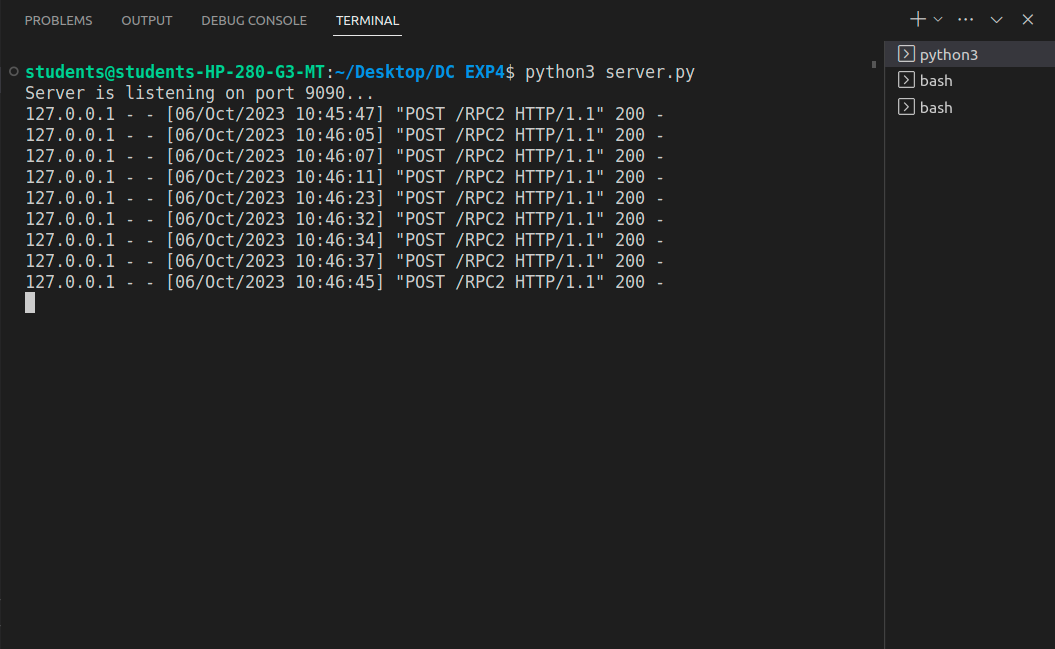
# Wait for the client thread to finish

client\_thread.join()

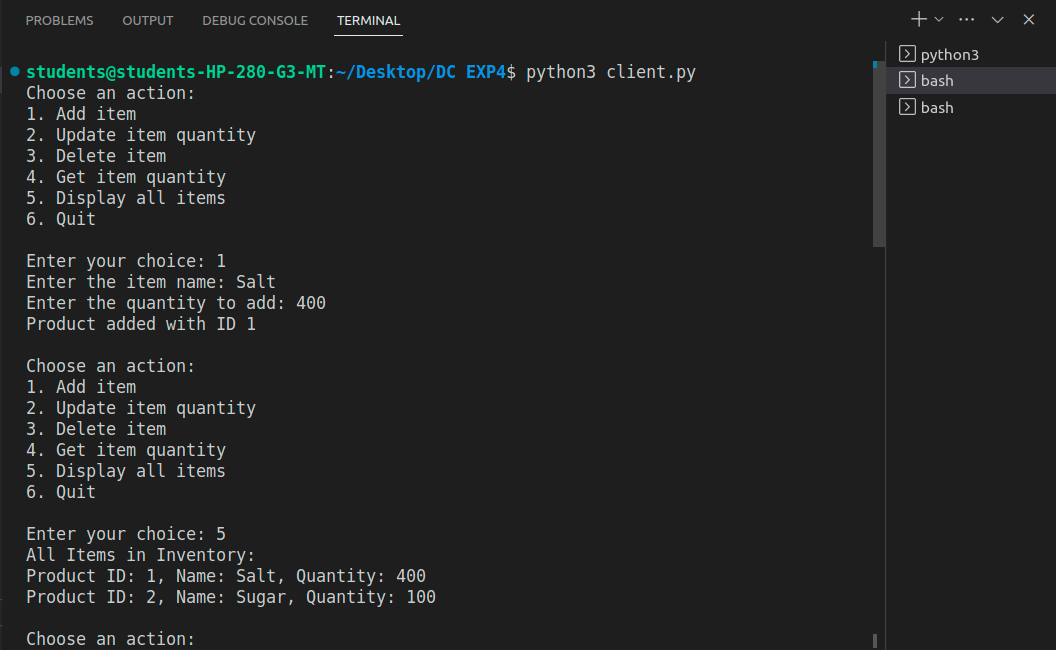
print("Client session ended.")

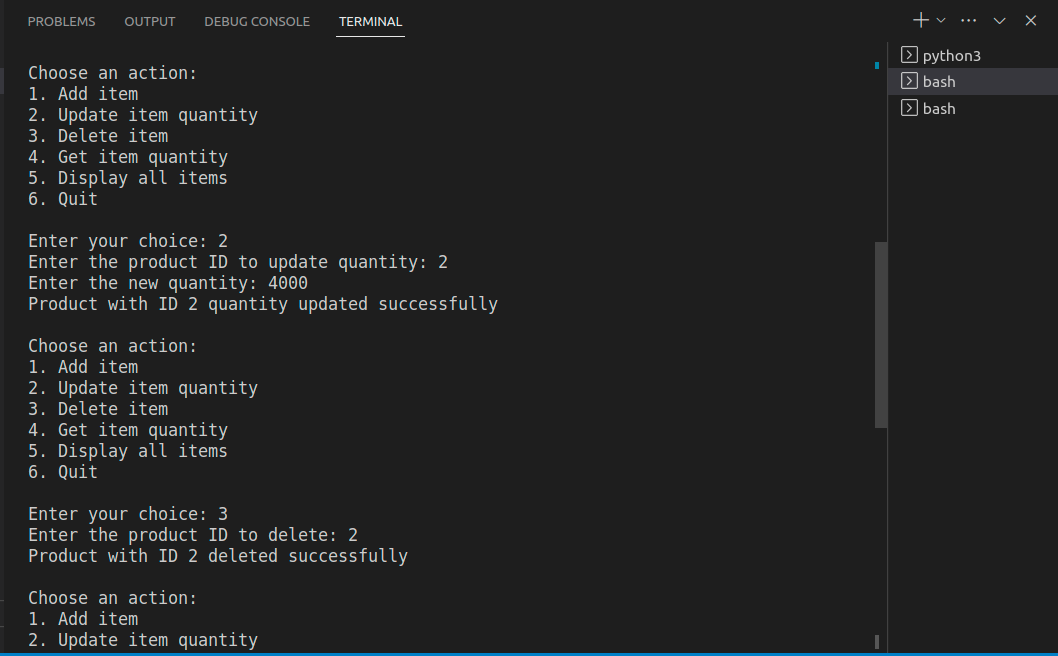
**Output:**

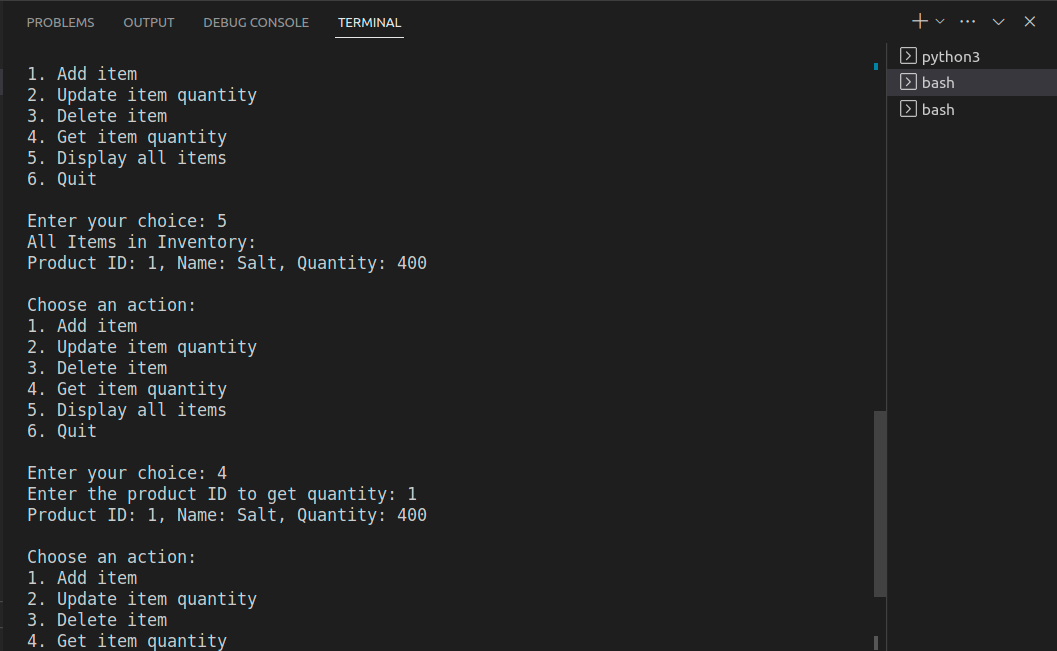
**Server:**

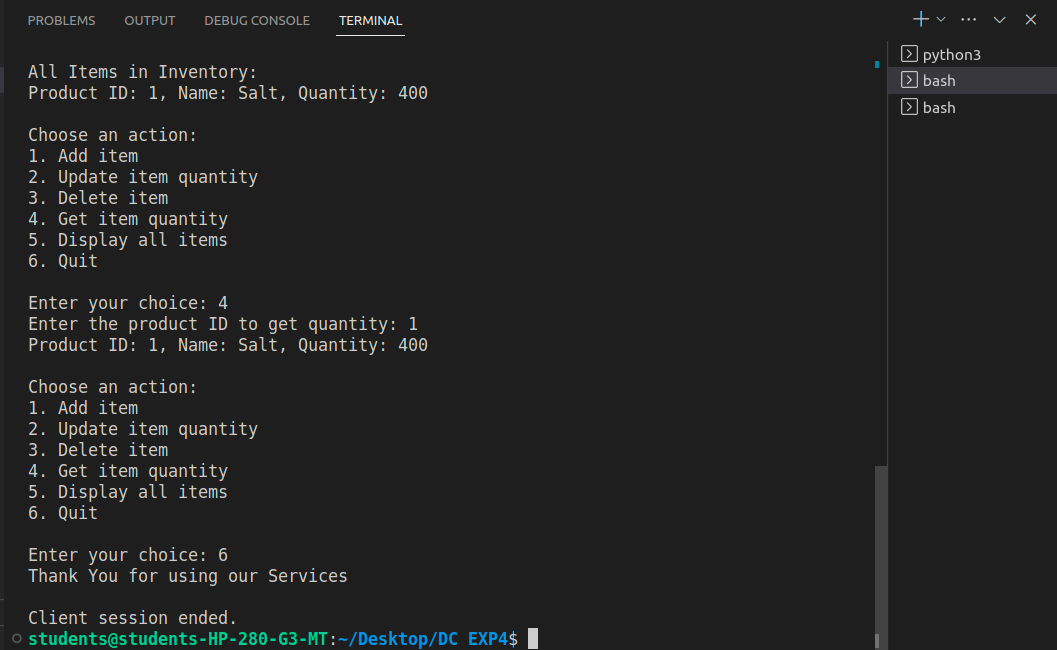
****

**Client 1:**

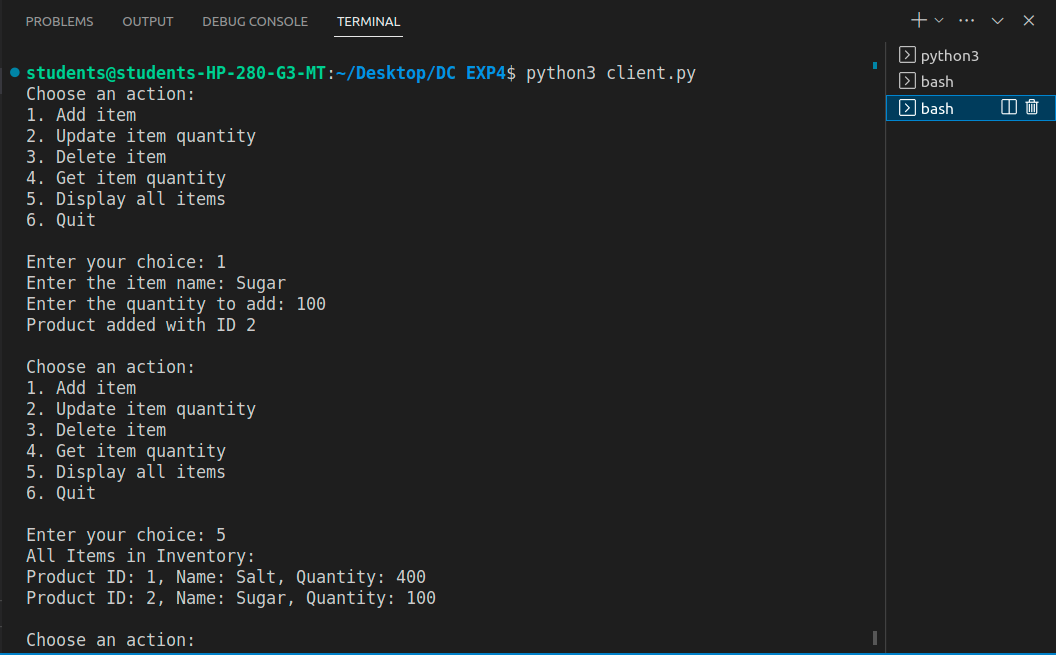
****

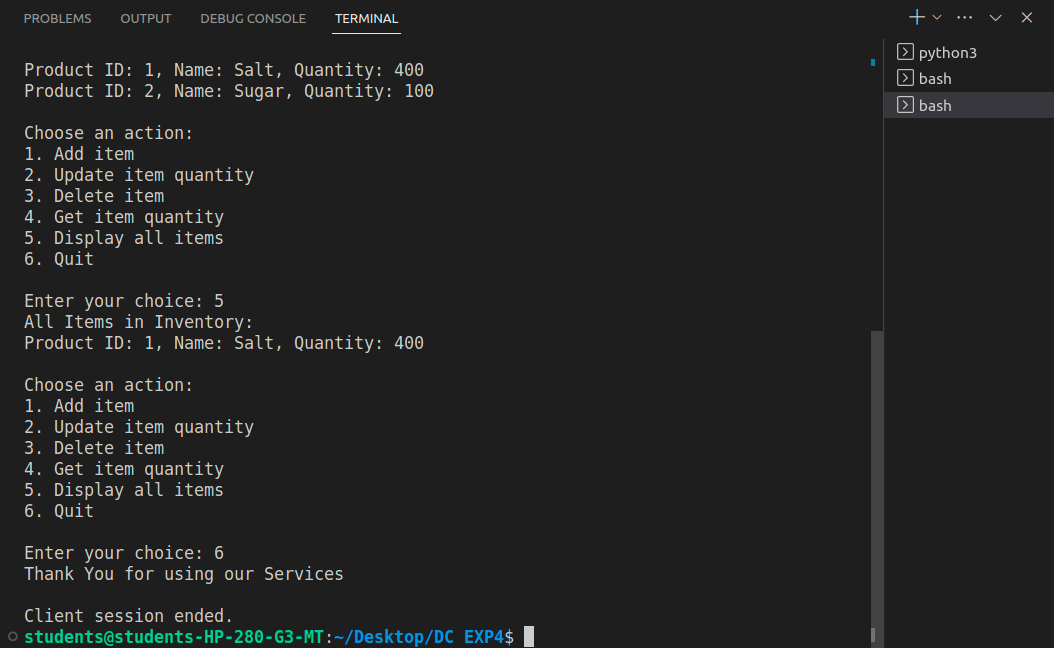
****

****

****

**Client 2:**

****

****

**Conclusion:**

Multi-threading in distributed systems using Python is a powerful technique for achieving parallelism, responsiveness, and efficient resource utilization. However, it requires careful consideration of concurrency control, fault tolerance, and load balancing. Implementing thread pools, robust error handling, and monitoring tools is crucial for success in building scalable and reliable distributed systems, ultimately leading to improved performance and a better user experience.