Roll No.

Date:

Experiment No. 3

Aim: To study and implement Group Communication.

Theory:

Group Communication in Distributed Systems

Group communication refers to the exchange of messages between multiple processes in a distributed system. It allows a process to send messages to multiple receivers efficiently, making it a core component of fault-tolerant, scalable, and distributed computing systems.

Group communication mechanisms ensure:

- Efficient message delivery: Reduces redundancy by sending messages to multiple recipients simultaneously.
- Fault tolerance: Supports replication and failover strategies.
- Consistency and ordering: Ensures messages are delivered in the correct order.

Types of Group Communication

Group communication can be classified based on the number of senders and receivers:

1. One-to-Many (1-M) Group Communication (Multicast Communication)

- In this model, a single sender transmits a message to multiple receivers.
- This is used in multicast communication, where one process wants to update or notify multiple processes at once.

Examples of 1-M Communication:

- Live streaming services (e.g., YouTube Live, Facebook Live).
- Software updates where a central server pushes updates to multiple nodes.
- Distributed database replication where updates must be propagated to all replicas.

Challenges & Solutions:

Challenge	Solution
Message Loss: Some receivers may not receive messages due to network failures.	Use acknowledgments and retransmission mechanisms (e.g., TCP-based multicast).
Scalability Issues: Sending messages to a large group can cause network congestion.	Use tree-based multicast protocols (e.g., IP Multicast, Gossip Protocols).
Ordering Problems: Different receivers may receive messages in different orders.	Implement event ordering techniques (absolute, consistent, causal ordering).

2. Many-to-One (M-1) Group Communication

- Multiple senders send messages to a single receiver.
- This is common in aggregation systems, where data from many sources is collected and processed by a single server.

Examples of M-1 Communication:

- IoT Systems: Multiple sensors send data to a central monitoring system.
- Distributed logging systems: Logs from multiple applications are sent to a central server.
- Client-server model: Multiple clients submitting queries to a single database server.

Challenges & Solutions:

Challenge	Solution
Message Overload: The receiver may get overwhelmed with too many messages.	Implement load balancing and message queuing (e.g., Kafka, RabbitMQ).
Message Ordering Issues: Messages from different senders may arrive in a different order.	Use sequence numbers or timestamps to order messages correctly.

3. Many-to-Many (M-N) Group Communication

- Multiple senders send messages to multiple receivers.
- This is the most complex group communication model and is used in peer-to-peer systems and collaborative applications.

Examples of M-N Communication:

- Multiplayer online games where multiple players send and receive updates.
- Blockchain networks where multiple nodes validate and share transaction data.
- Real-time collaboration tools like Google Docs.

Challenges & Solutions:

Challenge	Solution
Synchronization Issues: Messages must arrive at all recipients in the same order.	Use causal or total ordering algorithms (e.g., Vector Clocks, Lamport Timestamps).
Message Duplication: Some receivers may receive multiple copies of the same message.	Use deduplication mechanisms and message IDs.

Closed and Open Group Communication

A group in distributed systems can be classified based on who can send messages to it:

1. Closed Group Communication

- Only members of the group can send messages to the group.
- Restricts external processes from sending messages to ensure security.

Examples:

- A private Slack channel where only members can send and receive messages.
- A distributed database cluster where only authorized nodes can communicate.

2. Open Group Communication

- Any process (inside or outside the group) can send messages.
- Useful for public services where external clients interact with the system.

Examples:

- A public message board where anyone can post.
- Cloud services where external clients send requests to backend servers.

Comparison Table:

Feature	Closed Group Communication	Open Group Communication
Sender Restrictions	Only group members	External senders allowed
Security Level	Higher security	Lower security
Use Case	Private chat groups, distributed databases	Public forums, cloud APIs

Limited and Directed Broadcasting

Broadcasting refers to sending messages to multiple recipients.

1. Limited Broadcasting

- Messages are sent to a subset of nodes, not the entire network.
- Reduces network congestion.

Example:

- A university server sending notifications only to enrolled students.
- Corporate email lists sending messages to specific teams.

2. Directed Broadcasting

- Messages are sent to specific recipients using a routing mechanism.

- Avoids unnecessary transmissions.

Example:

- Targeted advertisements based on user preferences.
- Router forwarding rules sending packets to predefined devices.

Comparison Table:

Feature	Limited Broadcasting	Directed Broadcasting
Target Audience	Selected nodes	Specific recipients
Efficiency	Higher efficiency	Lower overhead
Example	University notifications	Router forwarding

Event Ordering

Event ordering ensures messages are delivered in a predictable and consistent sequence.

Types of Event Ordering

1. Absolute Ordering

- All processes receive messages in the exact same order.
- Uses timestamps or a central coordinator.

Example: Stock market transactions where updates must be processed in a strict order.

2. Consistent Ordering

All processes observe events in the same relative order, but timestamps may vary.

Example: Multiplayer gaming where all players see actions in the same sequence.

3. Causal Ordering

- If event A happened before B, then all processes must see A before B.
- Uses Lamport Timestamps or Vector Clocks.

Example: Messaging apps where replies must appear after the original message.

Code:

Absolute Ordering:

Client.py

```
import socket
import threading
class AbsoluteOrderingClient:
  def __init__(self,
server_ip="192.168.1.100", port=12345):
      self.client_socket =
socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
```

```
        self.client_socket.connect((server_ip,
port))
        threading.Thread(target=self.receive_
messages).start()
    def receive_messages(self):
        while True:
            try:
                message =
self.client_socket.recv(1024).decode()
                if not message:
                    break
                print(f"Ordered Message
Received: {message}")
            except:
                break
    def send_message(self, message):
        self.client_socket.send(message.enco
de())
if __name__ == "__main__":
    client =
AbsoluteOrderingClient(server_ip="192.1
68.3.244")
    while True:
        msg = input("Enter message: ")
        client.send_message(msg)
```

Server.py

```
import socket
import threading
class AbsoluteOrderingServer:
    def __init__(self, host="0.0.0.0",
port=12345):
        self.server_socket =
socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        self.server_socket.bind((host, port))
        self.server_socket.listen(5)
        self.clients = []
        self.message_queue = []
        self.message_counter = 0
        print(f"Server listening on
{host}:{port}")
    def broadcast(self, message):
        self.message_counter += 1
        ordered_message =
f"{self.message_counter}: {message}"
        self.message_queue.append(ordered_
message)
        for client in self.clients:
            client.send(ordered_message.encod
e())
    def handle_client(self, client_socket):
        while True:
            try:
                message =
client_socket.recv(1024).decode()
                if not message:
                    break
                print(f"Received: {message}")
                self.broadcast(message)
            except:
                break
        client_socket.close()
    def run(self):
        while True:
            client_socket, _ =
self.server_socket.accept()
            self.clients.append(client_socket)
            print("New client connected.")
            threading.Thread(target=self.handl
e_client, args=(client_socket,)).start()
if __name__ == "__main__":
    server = AbsoluteOrderingServer()
    server.run()
```

Consistent Ordering

Client.py

```
import socket
import threading
class ConsistentOrderingClient:
    def __init__(self,
server_ip="192.168.1.100", port=12345):
        self.client_socket =
socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        self.client_socket.connect((server_ip,
port))
        threading.Thread(target=self.receive_
messages).start()
```

```
    def receive_messages(self):
        while True:
            try:
                message =
self.client_socket.recv(1024).decode()
                if not message:
                    break
                print(f"Message Received:
{message}")
            except:
                break
    def send_message(self, message):
        self.client_socket.send(message.enco
de())
if __name__ == "__main__":
    client =
ConsistentOrderingClient(server_ip="192.
168.3.244")
    while True:
        msg = input("Enter message: ")
        client.send_message(msg)
```

Server.py

```
import socket
import threading
class ConsistentOrderingServer:
    def __init__(self, host="0.0.0.0",
port=12345):
        self.server_socket =
socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        self.server_socket.bind((host, port))
        self.server_socket.listen(5)
        self.clients = []
        print(f"Server listening on
{host}:{port}")
    def broadcast(self, message):
        for client in self.clients:
            client.send(message.encode())
    def handle_client(self, client_socket):
        while True:
            try:
                message =
client_socket.recv(1024).decode()
                if not message:
                    break
                print(f"Received: {message}")
                self.broadcast(message)
            except:
                break
        client_socket.close()
    def run(self):
        while True:
            client_socket, _ =
self.server_socket.accept()
            self.clients.append(client_socket)
            print("New client connected.")
            threading.Thread(target=self.handl
e_client, args=(client_socket,)).start()
if __name__ == "__main__":
    server = ConsistentOrderingServer()
    server.run()
```

Causal Ordering

Client.py

```
import socket
import threading
class CausalOrderingClient:
    def __init__(self,
server_ip="192.168.1.100", port=12345):
        self.client_socket =
socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        self.client_socket.connect((server_ip,
port))
        threading.Thread(target=self.receive_
messages).start()
    def receive_messages(self):
        while True:
            try:
                message =
self.client_socket.recv(1024).decode()
                if not message:
                    break
                print(f"Causal Ordered Message
Received: {message}")
            except:
                break
    def send_message(self, message):
        self.client_socket.send(message.encode())
if __name__ == "__main__":
    client = CausalOrderingClient(server_ip="192.168.3.244")
    while True:
        msg = input("Enter message: ")
        client.send_message(msg)
```

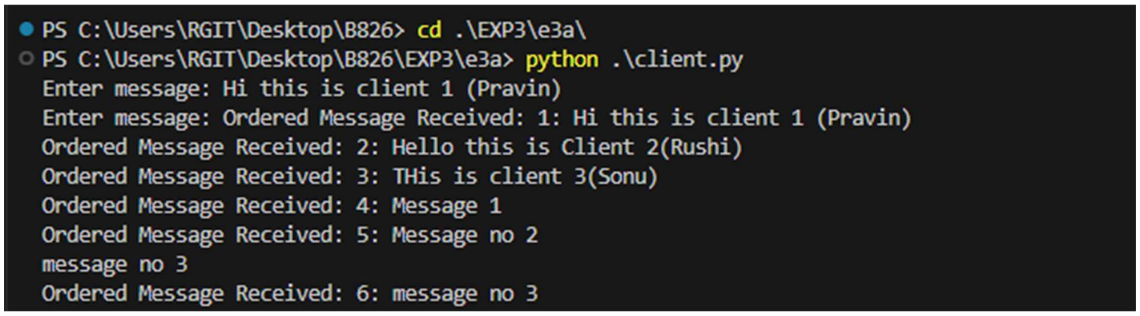
Server.py

```
import socket
import threading
class CausalOrderingServer:
    def __init__(self, host="0.0.0.0", port=12345):
        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server_socket.bind((host, port))
        self.server_socket.listen(5)
        self.clients = []
        self.history = []
        print(f"Server listening on {host}:{port}")
    def broadcast(self, message, sender):
        self.history.append((sender, message))
        for client in self.clients:
            if client != sender:
                client.send(message.encode())
    def handle_client(self, client_socket):
        while True:
            try:
                message = client_socket.recv(1024).decode()
                if not message:
                    break
                print(f"Received: {message}")
                self.broadcast(message, client_socket)
            except:
                break
        client_socket.close()
    def run(self):
        while True:
            client_socket, _ = self.server_socket.accept()
            self.clients.append(client_socket)
            print("New client connected.")
            threading.Thread(target=self.handle_client, args=(client_socket,)).start()
if __name__ == "__main__":
    server = CausalOrderingServer()
    server.run()
```

Output:

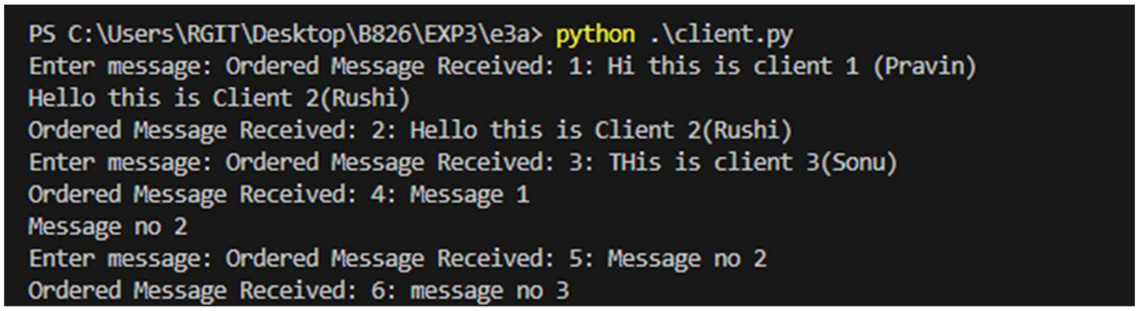
Absolute Ordering

Client1:



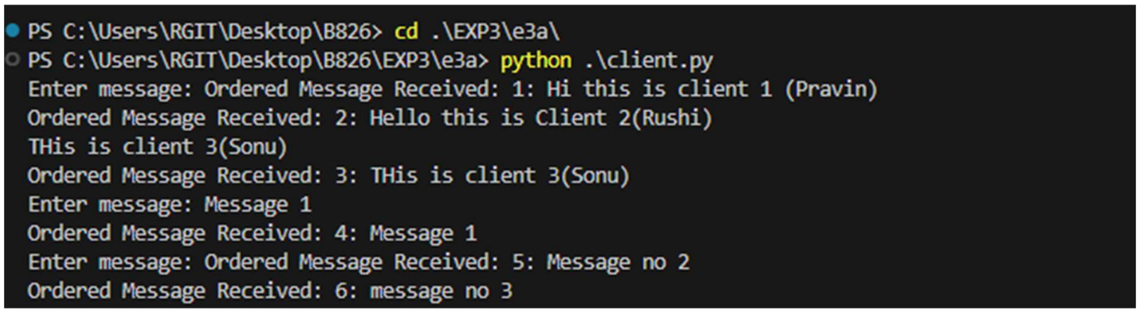
```
PS C:\Users\RGIT\Desktop\B826> cd .\EXP3\e3a\
PS C:\Users\RGIT\Desktop\B826\EXP3\e3a> python .\client.py
Enter message: Hi this is client 1 (Pravin)
Enter message: Ordered Message Received: 1: Hi this is client 1 (Pravin)
Ordered Message Received: 2: Hello this is Client 2(Rushi)
Ordered Message Received: 3: THis is client 3(Sonu)
Ordered Message Received: 4: Message 1
Ordered Message Received: 5: Message no 2
message no 3
Ordered Message Received: 6: message no 3
```

Client2:



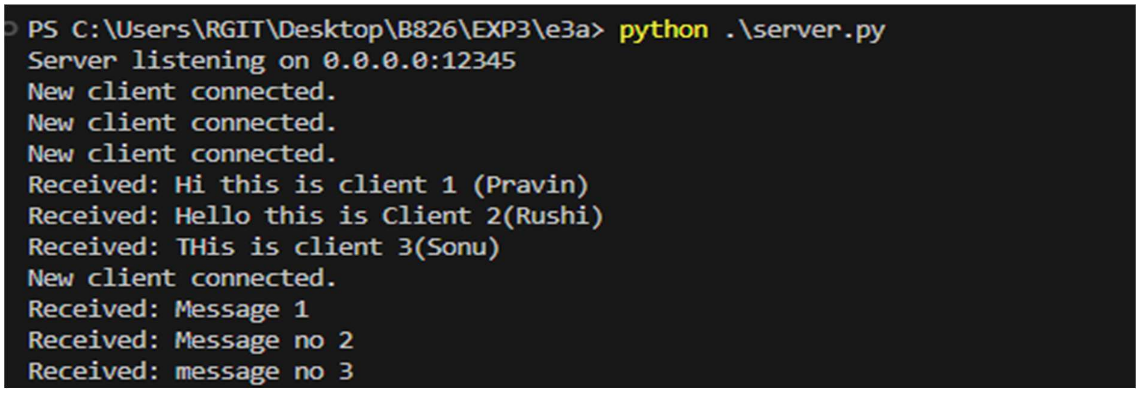
```
PS C:\Users\RGIT\Desktop\B826\EXP3\e3a> python .\client.py
Enter message: Ordered Message Received: 1: Hi this is client 1 (Pravin)
Hello this is Client 2(Rushi)
Ordered Message Received: 2: Hello this is Client 2(Rushi)
Enter message: Ordered Message Received: 3: THis is client 3(Sonu)
Ordered Message Received: 4: Message 1
Message no 2
Enter message: Ordered Message Received: 5: Message no 2
Ordered Message Received: 6: message no 3
```

Client3:



```
PS C:\Users\RGIT\Desktop\B826> cd .\EXP3\e3a\
PS C:\Users\RGIT\Desktop\B826\EXP3\e3a> python .\client.py
Enter message: Ordered Message Received: 1: Hi this is client 1 (Pravin)
Ordered Message Received: 2: Hello this is Client 2(Rushi)
THis is client 3(Sonu)
Ordered Message Received: 3: THis is client 3(Sonu)
Enter message: Message 1
Ordered Message Received: 4: Message 1
Enter message: Ordered Message Received: 5: Message no 2
Ordered Message Received: 6: message no 3
```

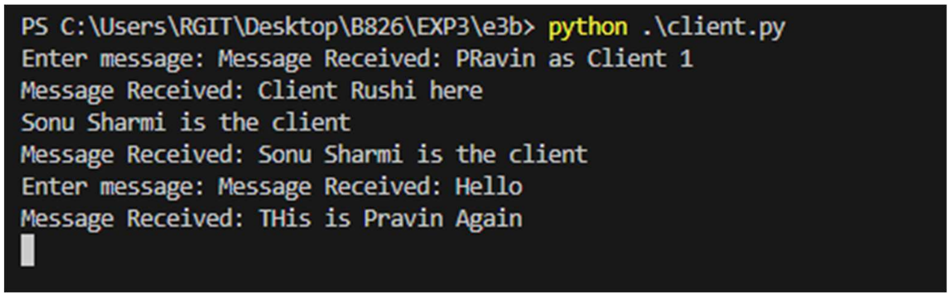
Server:



```
PS C:\Users\RGIT\Desktop\B826\EXP3\e3a> python .\server.py
Server listening on 0.0.0.0:12345
New client connected.
New client connected.
New client connected.
Received: Hi this is client 1 (Pravin)
Received: Hello this is Client 2(Rushi)
Received: THis is client 3(Sonu)
New client connected.
Received: Message 1
Received: Message no 2
Received: message no 3
```

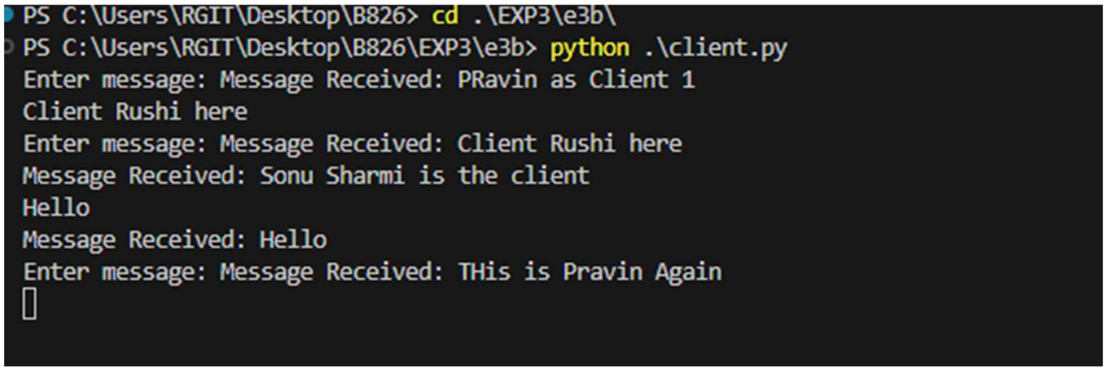
<u>Consistent Ordering</u>

Client1



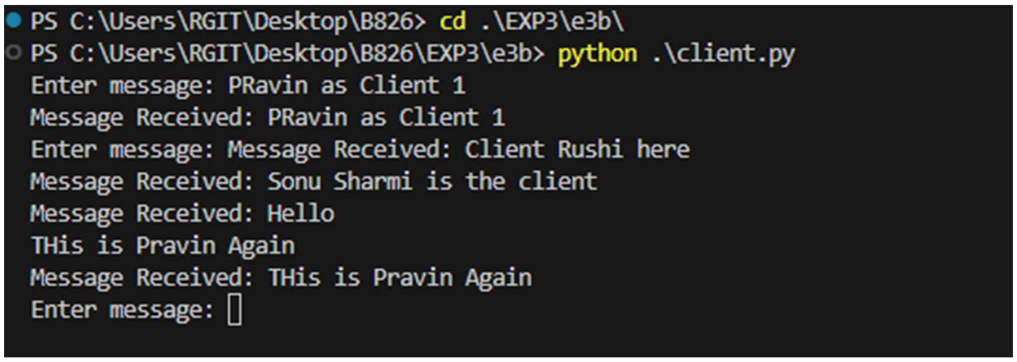
```
PS C:\Users\RGIT\Desktop\B826\EXP3\e3b> python .\client.py
Enter message: Message Received: PRavin as Client 1
Message Received: Client Rushi here
Sonu Sharmi is the client
Message Received: Sonu Sharmi is the client
Enter message: Message Received: Hello
Message Received: THis is Pravin Again
```

Client2



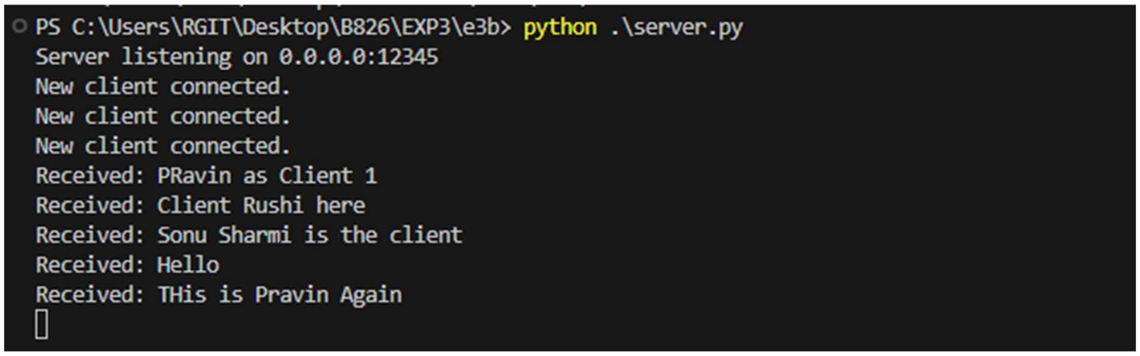
```
PS C:\Users\RGIT\Desktop\B826> cd .\EXP3\e3b\
PS C:\Users\RGIT\Desktop\B826\EXP3\e3b> python .\client.py
Enter message: Message Received: PRavin as Client 1
Client Rushi here
Enter message: Message Received: Client Rushi here
Message Received: Sonu Sharmi is the client
Hello
Message Received: Hello
Enter message: Message Received: THis is Pravin Again
```

Client3



```
PS C:\Users\RGIT\Desktop\B826> cd .\EXP3\e3b\
PS C:\Users\RGIT\Desktop\B826\EXP3\e3b> python .\client.py
Enter message: PRavin as Client 1
Message Received: PRavin as Client 1
Enter message: Message Received: Client Rushi here
Message Received: Sonu Sharmi is the client
Message Received: Hello
THis is Pravin Again
Message Received: THis is Pravin Again
Enter message:
```

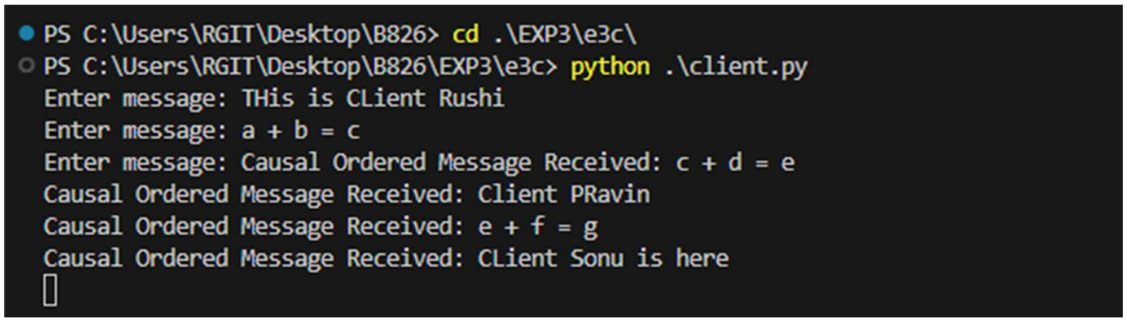
Server



```
PS C:\Users\RGIT\Desktop\B826\EXP3\e3b> python .\server.py
Server listening on 0.0.0.0:12345
New client connected.
New client connected.
New client connected.
Received: PRavin as Client 1
Received: Client Rushi here
Received: Sonu Sharmi is the client
Received: Hello
Received: THis is Pravin Again
```

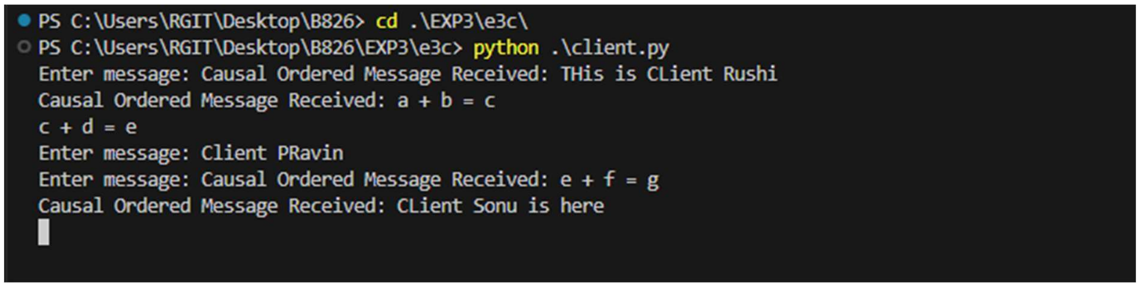
Causal Ordering

Client1:



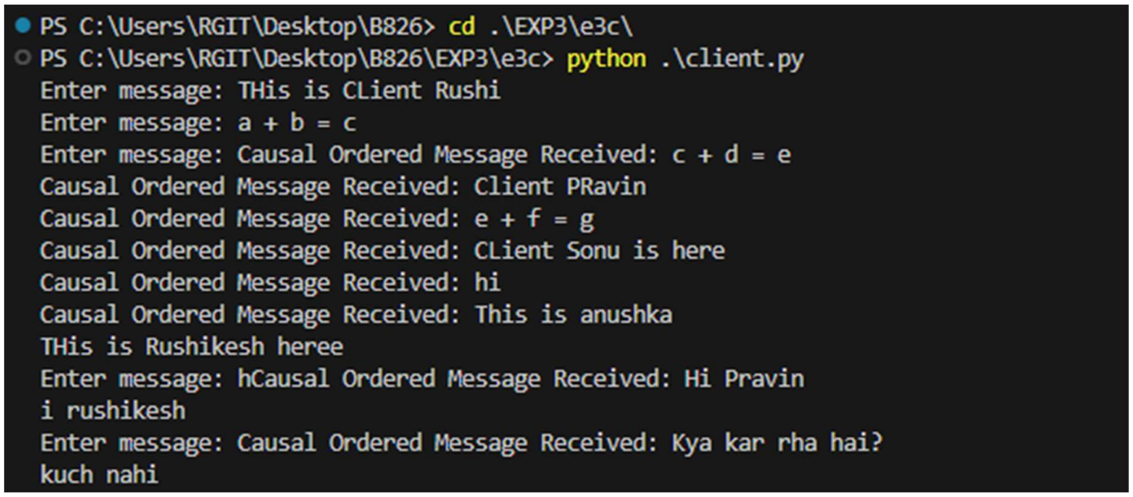
```
PS C:\Users\RGIT\Desktop\B826> cd .\EXP3\e3c\
PS C:\Users\RGIT\Desktop\B826\EXP3\e3c> python .\client.py
Enter message: THis is CLient Rushi
Enter message: a + b = c
Enter message: Causal Ordered Message Received: c + d = e
Causal Ordered Message Received: Client PRavin
Causal Ordered Message Received: e + f = g
Causal Ordered Message Received: CLient Sonu is here
```

Client2:



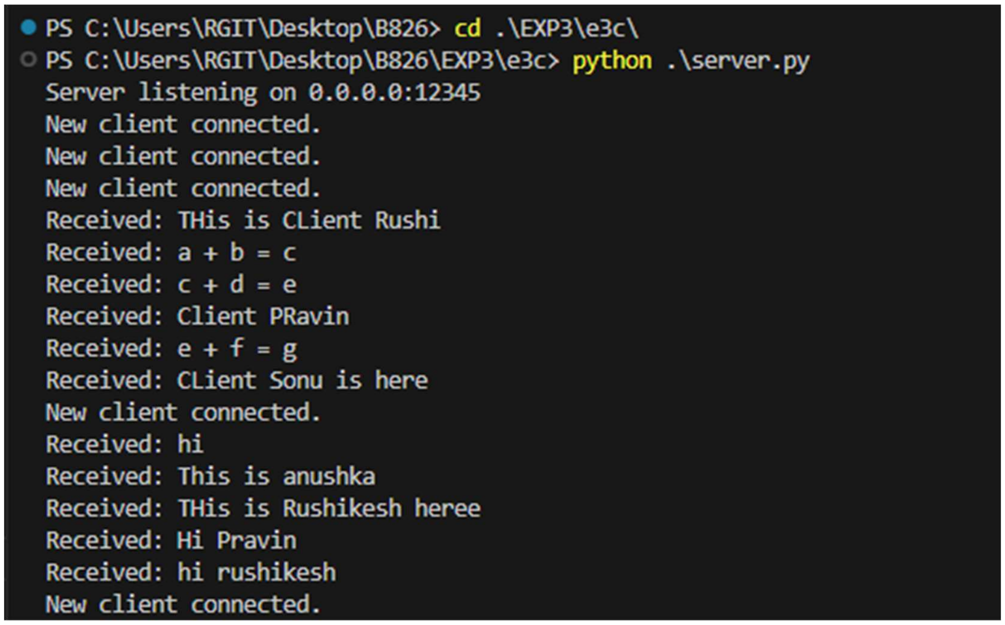
```
PS C:\Users\RGIT\Desktop\B826> cd .\EXP3\e3c\
PS C:\Users\RGIT\Desktop\B826\EXP3\e3c> python .\client.py
Enter message: Causal Ordered Message Received: THis is CLient Rushi
Causal Ordered Message Received: a + b = c
c + d = e
Enter message: Client PRavin
Enter message: Causal Ordered Message Received: e + f = g
Causal Ordered Message Received: CLient Sonu is here
```

Client3:



```
PS C:\Users\RGIT\Desktop\B826> cd .\EXP3\e3c\
PS C:\Users\RGIT\Desktop\B826\EXP3\e3c> python .\client.py
Enter message: THis is CLient Rushi
Enter message: a + b = c
Enter message: Causal Ordered Message Received: c + d = e
Causal Ordered Message Received: Client PRavin
Causal Ordered Message Received: e + f = g
Causal Ordered Message Received: CLient Sonu is here
Causal Ordered Message Received: hi
Causal Ordered Message Received: This is anushka
THis is Rushikesh heree
Enter message: hCausal Ordered Message Received: Hi Pravin
i rushikesh
Enter message: Causal Ordered Message Received: Kya kar rha hai?
kuch nahi
```

Server:



```
● PS C:\Users\RGIT\Desktop\B826> cd .\EXP3\e3c\
○ PS C:\Users\RGIT\Desktop\B826\EXP3\e3c> python .\server.py
  Server listening on 0.0.0.0:12345
  New client connected.
  New client connected.
  New client connected.
  Received: THis is CLient Rushi
  Received: a + b = c
  Received: c + d = e
  Received: Client PRavin
  Received: e + f = g
  Received: CLient Sonu is here
  New client connected.
  Received: hi
  Received: This is anushka
  Received: THis is Rushikesh heree
  Received: Hi Pravin
  Received: hi rushikesh
  New client connected.
```

Conclusion: We studied and implemented Group Communication in distributed systems, covering One-to-Many (1-M), Many-to-One (M-1), and Many-to-Many (M-N) communication models. We explored key challenges such as message loss, ordering issues, and scalability, along with their respective solutions like acknowledgment mechanisms, load balancing, and event ordering techniques. Additionally, we implemented and tested event ordering mechanisms including Absolute Ordering, Consistent Ordering, and Causal Ordering using Python socket programming.