**EXPERIMENT 3. Processes**

**Topic**: Implementing Threads for Parallel Processing

- **Objective**: Students will implement multi-threaded client-server communication, demonstrating process creation and management in a distributed system.

**Code**:

**Multi-threaded Server (Python)**:

```
import socket
import threading
def handle_client(client_socket):
    request = client_socket.recv(1024)
    print(f"Received: {request.decode()}")
    client_socket.send(b'Hello from server')
    client_socket.close()

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('localhost', 12345))
server.listen(5)

while True:
    client_sock, addr = server.accept()
    print(f"Connection from {addr}")
    client_thread = threading.Thread(target=handle_client, args=(client_sock,))
    client_thread.start()
```

**Client.py** (Python):

```
import socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(('localhost', 12345))
client_socket.send(b'Hello server')
response = client_socket.recv(1024)
print(f"Received from server: {response.decode()}")
client_socket.close()
```

**Steps**:

1. Run the multi-threaded server code.
2. Run the client and see how the server handles multiple clients using threads.

**EXPERIMENT 4. Communication**

**Topic**: Using Python for IPC via Sockets (Multicast Example)

- **Objective**: Demonstrate different communication techniques including Unicast, Multicast, and Broadcast.

**Code (Multicast)**:

**Multicast Server.py**:

```
import socket
```

```
import struct
MULTICAST_GROUP = '224.1.1.1'
PORT = 5007
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_socket.bind(('', PORT))

# Set up multicast group
group = socket.inet_aton(MULTICAST_GROUP)
server_socket.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, group +
socket.inet_aton('0.0.0.0'))

print(f"Multicast server running on {MULTICAST_GROUP}:{PORT}")
while True:
    data, addr = server_socket.recvfrom(1024)
    print(f"Received data: {data.decode()}")
```

**Multicast Client.py**:

```
import socket
import struct
MULTICAST_GROUP = '224.1.1.1'
PORT = 5007
client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
client_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
client_socket.bind(('', PORT))
group = socket.inet_aton(MULTICAST_GROUP)
client_socket.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, group +
socket.inet_aton('0.0.0.0'))

while True:
    data, addr = client_socket.recvfrom(1024)
    print(f"Received message: {data.decode()}")
```

**Steps**:

1. Run the server and client.
2. Observe how multicast communication works.

**EXPERIMENT 5. Naming**

**Topic**: Implementing Flat and Structured Naming

- **Objective**: Students will explore naming schemes in distributed systems using both flat and structured naming.

**Activity**:

- Students will implement a basic name resolution system using Python dictionaries (for flat naming) and hierarchical naming for structured naming.

**Code (Flat Naming)**:

```
# Flat Naming System
name_server = {
   "server1": "192.168.1.2",
   "server2": "192.168.1.3"
}

# Query
server = name_server.get("server1", None)
print(f"IP address of server1: {server}")
```

**Code (Structured Naming)**:

```
# Structured Naming System (Hierarchical)
name_server = {
   "region1": {
      "server1": "192.168.1.2",
      "server2": "192.168.1.3"
   },
   "region2": {
      "server3": "192.168.2.1",
      "server4": "192.168.2.2"
   }
}

# Query
server = name_server["region1"].get("server1", None)
print(f"IP address of region1 server1: {server}")
```

**EXPERIMENT 6. The Synchronization Process**

**Topic**: Implementing a Simple Synchronization Algorithm (Lamport's Logical Clocks)

- **Objective**: Implement the Lamport's logical clock algorithm to ensure proper synchronization in a distributed system.

**Code**:

```
# Lamport's Logical Clock
class LamportClock:
   def __init__(self):
      self.time = 0

   def send_message(self):
      self.time += 1
      return self.time

   def receive_message(self, received_time):
      self.time = max(self.time, received_time) + 1
      return self.time
```

```
# Example usage
clock1 = LamportClock()
clock2 = LamportClock()

# Simulate messages
message_time1 = clock1.send_message()
print(f"Clock1 Time after sending: {message_time1}")

message_time2 = clock2.receive_message(message_time1)
print(f"Clock2 Time after receiving: {message_time2}")
```