EXPERIMENT 7: IPC Using Python Sockets (Client-Server Communication)
Objective: Understand inter-process communication using TCP sockets.

Code
Server (server.py):

```python
import socket  # Import the socket module

# Create a TCP/IP socket
s = socket.socket()

# Bind the socket to a specific host and port
s.bind(('localhost', 12345))

# Listen for incoming connections (max 1 connection waiting in queue)
s.listen(1)
print("Server listening...")

# Accept a connection from a client
conn, addr = s.accept()
print("Connected by", addr)

# Receive up to 1024 bytes of data from the client
data = conn.recv(1024)
print("Received:", data.decode())  # Decode and print the received message

# Send a response back to the client
conn.send(b'Hello from server')

# Close the connection
conn.close()
```

Client (client.py):

```python
import socket  # Import the socket module

# Create a TCP/IP socket
s = socket.socket()

# Connect to the server running on localhost at port 12345
s.connect(('localhost', 12345))

# Send a message to the server
s.send(b'Hello from client')

# Receive the server's response (up to 1024 bytes)
data = s.recv(1024)
print("Received:", data.decode())  # Decode and print the server's reply

# Close the connection
s.close()
```

Tasks

1. Add exception handling using try/except.
2. Use threading on the server to handle multiple clients.
3. Implement basic encryption (e.g., using base64).

EXPERIMENT 8: Clock Synchronization Simulation (Berkeley Algorithm)

Objective: Simulate a simple form of clock synchronization.

Code

```python
import random  # Import the random module (not used in this snippet but often useful in simulations)

# Function to apply the Berkeley Algorithm for clock synchronization
def berkeley_algorithm(clocks):
    # Calculate the average time of all clocks
    avg_time = sum(clocks) / len(clocks)

    # Calculate the adjustment needed for each clock
    # Adjustment = average time - current clock time
    adjustments = [round(avg_time - c, 2) for c in clocks]

    return adjustments

# Simulated clock times (in seconds)
clocks = [100.3, 102.5, 98.4, 101.0]

# Get adjustments using the Berkeley algorithm
adjustments = berkeley_algorithm(clocks)

# Display the original clock values
print("Original Clocks:", clocks)

# Show the calculated adjustments for each clock
print("Adjustments:", adjustments)

# Apply the adjustments to synchronize all clocks
synchronized = [round(clocks[i] + adjustments[i], 2) for i in range(len(clocks))]
print("Synchronized Clocks:", synchronized)


```

Output

Output

```
Original Clocks: [100.3, 102.5, 98.4, 101.0]
Adjustments: [0.25, -1.95, 2.15, -0.45]
Synchronized Clocks: [100.55, 100.55, 100.55, 100.55]
```

Tasks

1. Add simulated network delays.
2. Introduce a leader election (select lowest clock node).
3. Convert to a multi-process simulation with real-time syncing.

EXPERIMENT 9: Mutual Exclusion Using Ricart-Agrawala Algorithm (Simulation)

Objective: Understand message-based mutual exclusion.

Code

```python
# Define a class to represent a node in a distributed system
class Node:
    def __init__(self, id):
        self.id = id                    # Unique identifier for the node
        self.request_queue = []        # Queue to track requests (not used in this simple example)

    # Simulate a request to enter the critical section
    def request_cs(self, timestamp):
        print(f"Node {self.id} requests CS at time {timestamp}")
        return f"REQ from {self.id} at {timestamp}"  # Return a request message as a string

    # Simulate receiving a reply from another node
    def receive_reply(self, from_id):
        print(f"Node {self.id} received REPLY from {from_id}")

# ---------------- Simulation Starts Here ----------------

# Create two nodes
node1 = Node(1)
node2 = Node(2)

# Both nodes request to enter the critical section with different timestamps
req1 = node1.request_cs(5)
req2 = node2.request_cs(4)

# Decide which node gets access based on the request message (lexicographic comparison)
# NOTE: This is a simplified comparison; in real systems, timestamps would be compared directly
if req1 > req2:
    node2.receive_reply(1)  # Node 2 gets the CS, Node 1 sends REPLY
else:
    node1.receive_reply(2)  # Node 1 gets the CS, Node 2 sends REPLY

```

Output

```
Output

Node 1 requests CS at time 5
Node 2 requests CS at time 4
Node 1 received REPLY from 2
```

Tasks

1. Use sockets to simulate message passing.
2. Add delays and simulate failure handling.

3. Implement a queue to track deferred replies.