Experiment 05- Develop a program for Election Algorithm

Learning Objective: To understand the concept of election algorithms in distributed computing and to develop a program that demonstrates the election process among distributed nodes.

Introduction:

Introduction to Election Algorithms

Election algorithms are used in distributed systems to ensure that a single process is designated as the coordinator or leader among a group of processes. These algorithms are crucial in scenarios where a centralized authority is required to manage system resources efficiently.

Key Characteristics of Election Algorithms:

1. **Fault Tolerance:** The ability to recover from node failures and elect a new leader.
2. **Fairness:** Every node should have a fair chance to be elected.
3. **Scalability:** The algorithm should perform efficiently in large-scale distributed systems.
4. **Message Complexity:** The number of messages exchanged should be minimal to ensure efficiency.

Types of Election Algorithms:

1. **Bully Algorithm:**
   - Initiated when a node detects a failure of the current leader.
   - The node with the highest process ID takes over as the leader.
   - Requires multiple message exchanges to determine the leader.
2. **Ring Algorithm:**
   - Processes are arranged in a logical ring structure.
   - A token is passed around the ring to determine the highest ID process.
   - The node with the highest ID is elected as the leader.
3. **Modified Election Algorithms:**
   - Hybrid approaches that combine aspects of the Bully and Ring algorithms.
   - Reduce message complexity and improve fault tolerance.

Use Cases of Election Algorithms:

- Distributed database management.
- Cloud computing for resource coordination.
- Load balancing in network systems.

Implementation:

Welcome | server.py | client.py

server.py > handle_client

```python
import socket
import threading

# Server configuration
HOST = '127.0.0.1'
PORT = 5000

# List to store connected clients
clients = []

# Function to handle incoming messages from a client
# Each client's thread keeps running until they disconnect.
def handle_client(client_socket, address):
    print(f"[NEW CONNECTION] {address} connected.")
    while True:  # Keep listening for messages
        try:
            message = client_socket.recv(1024).decode('utf-8') # Receive data (1024 bytes max)
            if not message:      # If message is empty (Client disconnected manually), client disconnected
                break
            print(f"[{address}] {message}")
            formatted_message = f"[{address}] {message}"
            broadcast(formatted_message, client_socket)  # Send the message to other clients
        except:          # If an error occurs (e.g., client disconnects due to lost internet connection)
            clients.remove(client_socket)   # Remove the disconnected client from the list
            client_socket.close()           # Close the client connection
            break
```

server.py > handle_client

```python

# Function to broadcast a message to all clients except the sender
def broadcast(message, sender_socket):
    for client in clients:
        if client != sender_socket:   # Don't send the message back to the sender
            try:
                client.send(message.encode('utf-8'))
            except:
                clients.remove(client)  # Remove client if sending fails (e.g., they disconnected)

# Main function to start the server
def start_server():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # Create a socket for IPv4 (AF_INET) & TCP (SOCK_STREAM)
    server.bind((HOST, PORT))  # Bind the server to the IP and port
    server.listen()  # Start listening for incoming connections
    print(f"Server started on {HOST}:{PORT}")  # Print message that server is running

    while True:  # Keep accepting new client connections
        client_socket, client_address = server.accept()  # Accept a new client connection
        clients.append(client_socket)  # Add the new client to the list
        thread = threading.Thread(target=handle_client, args=(client_socket, client_address))  # Create a new thread for each client
        thread.start()  # Start the thread to handle the client

# Run the server
if __name__ == "__main__":
    start_server()
```

Result and Discussion:

```
Process 87 announces leadership to Process 56
PS C:\Users\Urmil Pawar\Documents\Sem 8\DC\Election algorithm> python ring.py
Leader 87 has failed!
Process 56 starts an election.
Process 56 → sends election message to → 87
Process 87 → sends election message to → 1
Process 1 → sends election message to → 3
Process 3 → sends election message to → 5
Process 5 → sends election message to → 7
Process 7 → sends election message to → 9
Process 9 → sends election message to → 10
Process 10 → sends election message to → 12
Process 12 → sends election message to → 24

New Leader Elected: 87
Process 87 announces leadership to Process 1
Process 87 announces leadership to Process 3
Process 87 announces leadership to Process 5
Process 87 announces leadership to Process 7
Process 87 announces leadership to Process 9
Process 87 announces leadership to Process 10
Process 87 announces leadership to Process 12
Process 87 announces leadership to Process 24
Process 87 announces leadership to Process 56
PS C:\Users\Urmil Pawar\Documents\Sem 8\DC\Election algorithm>
```

Learning Outcomes: The student should have the ability to:

- **LO1 :** Understanding of leader election in distributed computing.
- **LO2 :** Practical experience in implementing election algorithms.
- **LO3 :** Ability to analyze and optimize leader election mechanisms.

Course Outcomes: Upon completion of the course, students will be able to understand and implement different types of Election Algorithms.

Conclusion:

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	Total
Marks Obtained				