Exercise1: Inter-Process Communication in Distributed Computing

Objective:

- To understand and implement inter-process communication using Java sockets.
- To explore and implement Remote Method Invocation (RMI) for distributed computing.

Task 1: Build a Simple Client-Server Application

Description: Create a simple client-server application where the server provides a service and the client communicates with it.

Task 2: Enhanced Communication with Data Transfer

Description: Modify the client-server application to send and receive user-defined objects using ObjectInputStream and ObjectOutputStream.

Part 2: Remote Method Invocation (RMI)

Task 1: Create an RMI Service

Description: Develop an RMI-based distributed application where the server provides a remote service, and the client invokes this service.

Solution:

Part 1: Inter-Process Communication using Python Sockets

Task 1: Simple Client-Server Application

Server Code (Python):

```python
import socket

def simple_server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('localhost', 5000))
    server_socket.listen(1)

    print("Server is listening on port 5000...")

    connection, address = server_socket.accept()
    print(f"Connection from {address} has been established.")

    client_message = connection.recv(1024).decode('utf-8')
    print(f"Client says: {client_message}")

    server_message = "Hello from the server!"
    connection.sendall(server_message.encode('utf-8'))

    connection.close()
    server_socket.close()
```

```python
if __name__ == "__main__":
    simple_server()
```

Client Code (Python):

```python
import socket

def simple_client():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect(('localhost', 5000))

    client_message = "Hello from the client!"
    client_socket.sendall(client_message.encode('utf-8'))

    server_message = client_socket.recv(1024).decode('utf-8')
    print(f"Server says: {server_message}")

    client_socket.close()

if __name__ == "__main__":
    simple_client()
```

Task 2: Enhanced Communication with Data Transfer (Sending and Receiving Objects)

In this task, we modify the client-server application to send and receive custom objects using Python's pickle module, which allows serialization of objects.

Server Code (Python with Object Transfer):

```python
import socket
import pickle

class CustomObject:
    def __init__(self, name, value):
        self.name = name
        self.value = value

def enhanced_server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('localhost', 5000))
    server_socket.listen(1)

    print("Server is listening on port 5000...")

    connection, address = server_socket.accept()
    print(f"Connection from {address} has been established.")

    # Receive object
    data = connection.recv(4096)
    received_object = pickle.loads(data)
```

```python
    print(f"Received object: Name = {received_object.name}, Value =
{received_object.value}")

    # Send response object
    response_object = CustomObject("ResponseObject", 123)
    connection.sendall(pickle.dumps(response_object))

    connection.close()
    server_socket.close()

if __name__ == "__main__":
    enhanced_server()
```

Client Code (Python with Object Transfer):

```python
import socket
import pickle

class CustomObject:
    def __init__(self, name, value):
        self.name = name
        self.value = value

def enhanced_client():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect(('localhost', 5000))

    # Create and send custom object
    client_object = CustomObject("ClientObject", 42)
    client_socket.sendall(pickle.dumps(client_object))

    # Receive response object
    data = client_socket.recv(4096)
    response_object = pickle.loads(data)
    print(f"Server response: Name = {response_object.name}, Value =
{response_object.value}")

    client_socket.close()

if __name__ == "__main__":
    enhanced_client()
```

Part 2: Remote Method Invocation (RMI)

Although Python doesn't have a native RMI system like Java, the equivalent would be using **XML-RPC** or **Pyro4** to implement remote method invocation. Below is an example using **Pyro4**.

First, you need to install Pyro4:

pip install Pyro4

Task 1: Create an RMI Service

RMI Server Code (Python using Pyro4):

```
import Pyro4

# Create a class for the remote service
@Pyro4.expose
class RemoteService:
    def say_hello(self, name):
        return f"Hello, {name}! This is the RMI server."

def start_server():
    # Start the Pyro4 Daemon
    daemon = Pyro4.Daemon()  # Pyro4 daemon
    ns = Pyro4.locateNS()  # Locate the name server
    uri = daemon.register(RemoteService)  # Register the service
    ns.register("example.rmi", uri)  # Register the service with a name in the
name server

    print(f"RMI Server started with URI: {uri}")
    daemon.requestLoop()  # Start the event loop for listening

if __name__ == "__main__":
    start_server()
```

RMI Client Code (Python using Pyro4):

```
import Pyro4

def rmi_client():
    # Locate the RMI server by the name 'example.rmi'
    remote_service = Pyro4.Proxy("PYRONAME:example.rmi")

    # Call the remote method
    response = remote_service.say_hello("Client")
    print(f"Server says: {response}")

if __name__ == "__main__":
    rmi_client()
```

Running the Pyro4-based RMI Application:

1. Start the name server by running:

   python -m Pyro4.naming

2. Run the server code (RMI Server), which will register the service and wait for requests.

3. Run the client code (RMI Client), which will connect to the server and invoke the remote method.

Exercise 3: Message Passing and Distributed Mutual Exclusion

Objective:

To understand and implement message passing between distributed processes.

To implement a distributed mutual exclusion algorithm to manage resource access in a distributed system.

Part 1: Message Passing

Task 1: Implement a Simple Message Passing System

Description: Create a message passing system where multiple processes communicate with each other using Java sockets. Each process will send and receive messages.

Part 2: Distributed Mutual Exclusion

Task 1: Implement Distributed Mutual Exclusion Using Ricart-Agrawala Algorithm

Description: Implement a distributed mutual exclusion algorithm (Ricart-Agrawala) to ensure that only one process at a time can access a critical section.

Solution:

Part 1 – Message Passing

Server Code

```
import socket
import threading
import sys

clients=[]

def handle_clients(client_socket,client_address):

    print(f"Client {client_address} has connected to the server")

    clients.append(client_socket)

    while True:
        try:
         message=client_socket.recv(1024).decode()

         if message:
            broadcast(client_socket,message)
        except:
           break

    print("Disconnected from connection")

    clients.remove(client_socket)
    client_socket.close()
```

```
def broadcast(client_socket,message):
   for client in clients:
      try:
        if client!=client_socket:
            client.send(message.encode('utf-8'))
      except:
         clients.remove(client)
         client.close()

server_socket=socket.socket(socket.AF_INET6,socket.SOCK_STREAM)

host="localhost"
port=5002

server_socket.bind((host,port))

server_socket.listen()

while True:
   client_socket,addr=server_socket.accept()
   thread=threading.Thread(target=handle_clients,args=(client_socket,addr))
   thread.start()
```

Client:

```
import threading
import socket
import sys

lock=threading.Lock()

def receive_messages(client_socket):

    while True:
        try:
                message=client_socket.recv(1024).decode()
                if message:
                    with lock:
                        sys.stdout.write("\r\033[K")

                        print(f"Message from Server:{message}")

                        print("YOU:",end='',flush=True)
        except:
            client_socket.close()
            break


client_socket=socket.socket(socket.AF_INET6,socket.SOCK_STREAM)

host="localhost"
```

```python
port=5002

client_socket.connect((host,port))

thread=threading.Thread(target=receive_messages,args=(client_socket,))
thread.start()

while True:
    try:
        message=input("YOU:")
        if message.lower()=="exit":
            client_socket.send("Client is exiting".encode('utf-8'))
            client_socket.close()
            break
        else:
            client_socket.send(message.encode('utf-8'))
    except:
        client_socket.close()
        break
```

Part 2: Distributed Mutual Exclusion

```python
import threading
import time
import random

from queue import PriorityQueue

class RicartAgrawala:
    def _init_(self, pid, total_processes):
        self.pid = pid
        self.processes = total_processes
        self.lock = threading.Lock()
        self.reply_count = 0
        self.requesting_CS = False
        self.timestamp = 0
        self.request_queue = PriorityQueue()

    def request_cs(self):
        self.requesting_CS = True
        self.timestamp += 1
        print(f"Process {self.pid} requests to enter the CS")
        for i in range(self.processes):
            if self.pid != i:
                self.send_message(i)
        while self.reply_count < self.processes - 1:
            time.sleep(1)
        self.enter_cs()

    def send_message(self, target_pid):
        print(f"Process {self.pid} sends message to {target_pid} at timestamp {self.timestamp}")
```

```
        processes[target_pid].receive_message(self)

    def receive_message(self, sender):
        with self.lock:
            self.timestamp = max(self.timestamp, sender.timestamp) + 1
            print(f"Process {self.pid} received message from {sender.pid}")
            if not self.requesting_CS or (sender.timestamp, sender.pid) <
(self.timestamp, self.pid):
                self.send_reply(sender)
            else:
                self.request_queue.put((sender.timestamp, sender))

    def send_reply(self, recipient):
        print(f"Process {self.pid} replies to {recipient.pid}")
        recipient.receive_reply(self)

    def receive_reply(self, _):
        self.reply_count += 1

    def enter_cs(self):
        print(f"Process {self.pid} enters the CS at timestamp
{self.timestamp}")
        time.sleep(random.uniform(0.5, 1.5))
        self.exit_cs()

    def exit_cs(self):
        self.reply_count = 0
        self.requesting_CS = False
        print(f"Process {self.pid} exits the CS at timestamp
{self.timestamp}")
        while not self.request_queue.empty():
            _, process = self.request_queue.get()
            self.send_reply(process)

def process_action(process):
    process.request_cs()
    time.sleep(random.uniform(0.5, 1.5))

if _name_ == "_main_":
    num_processes = 5
    processes = [RicartAgrawala(i, num_processes) for i in
range(num_processes)]
    threads = [threading.Thread(target=process_action, args=(p,)) for p in
processes]

    for thread in threads:
        thread.start()
    for thread in threads:
        thread.join()
```

Exercise 4: Clock and Time Synchronization in Distributed Computing

Objective:

To understand and implement logical clocks and vector clocks in distributed systems.

To explore how clock synchronization helps maintain consistent order of events in distributed systems.

Part 1: Logical Clocks

Task 1: Implement Logical Clocks Using Lamport's Logical Clock

Description: Implement a simple distributed system with multiple processes that use Lamport's Logical Clock to order events. Each process will increment its logical clock and send time-stamped messages to other processes.

Part 2: Vector Clocks

Task 1: Implement Vector Clocks

Description: Implement vector clocks to handle more complex scenarios where multiple processes can be in concurrent states. Vector clocks help in determining the partial ordering of events and detecting causality.

Solution:

Part 1: Logical Clocks

```
import threading
import time
import random

class LogicalClock:
    def __init__(self, pid):
        self.pid = pid  # Process ID
        self.clock = 0  # Logical clock value
        self.lock = threading.Lock()  # Lock for thread safety

    def increment_clock(self):
        """ Increment the clock for this process. """
        self.clock += 1

    def send_message(self, receiver):
        """ Send a message to another process with the current logical clock. """
        with self.lock:
            self.increment_clock()
```

```
            print(f"Process {self.pid} sending message with timestamp {self.clock} to
Process {receiver.pid}")
        receiver.receive_message(self.clock)

    def receive_message(self, received_time):
        """ Update the logical clock on receiving a message. """
        with self.lock:
            self.clock = max(self.clock, received_time) + 1
            print(f"Process {self.pid} received message with timestamp
{received_time}. Updated clock to {self.clock}")

    def perform_event(self):
        """ Simulate an internal event that increments the logical clock. """
        with self.lock:
            self.increment_clock()
            print(f"Process {self.pid} performs an internal event. Updated clock to
{self.clock}")

    def run(self, peers):
        """ Simulate sending and receiving messages between processes. """
        for _ in range(5):
            # Perform an internal event
            self.perform_event()
            time.sleep(random.random())  # Random sleep to simulate concurrency

            receiver = random.choice(peers)  # Randomly pick a process to send a
message to
            if receiver != self:  # Don't send to itself
                self.send_message(receiver)
            time.sleep(random.random())  # Random sleep to simulate concurrency


# Simulating logical clock across multiple processes using threading
def start_simulation():
    num_processes = 3
    processes = [LogicalClock(i) for i in range(1, num_processes + 1)]

    threads = []
    for process in processes:
        thread = threading.Thread(target=process.run, args=(processes,))
        threads.append(thread)
        thread.start()

    # Wait for all threads to complete
    for thread in threads:
        thread.join()
```

```
start_simulation()
```

Part 2 – Vector Clocks

```
import threading
import time
import random

class VectorClock:
    def __init__(self, pid, num_processes):
        self.pid = pid  # Process ID
        self.clock = [0] * num_processes  # Vector clock
        self.num_processes = num_processes
        self.lock = threading.Lock()  # Lock for thread safety

    def increment_clock(self):
        """ Increment the clock for this process. """
        self.clock[self.pid] += 1

    def send_message(self, receiver, message):
        """ Send a message to another process with the current vector clock. """
        with self.lock:
            self.increment_clock()
            print(f"Process {self.pid} sending message '{message}' with clock {self.clock} to 
Process {receiver.pid}")
        receiver.receive_message(message, self.clock.copy())

    def receive_message(self, message, sender_clock):
        """ Update the vector clock on receiving a message. """
        with self.lock:
            # Element-wise max between the current clock and the received clock
            for i in range(self.num_processes):
                self.clock[i] = max(self.clock[i], sender_clock[i])
            self.increment_clock()  # Increment own clock after receiving the message
            print(f"Process {self.pid} received message '{message}' with sender's clock 
{sender_clock}. Updated clock to {self.clock}")

    def perform_event(self):
        """ Simulate an internal event that increments the vector clock. """
        with self.lock:
            self.increment_clock()
            print(f"Process {self.pid} performs an internal event. Updated clock to 
{self.clock}")

    def run(self, peers):
        """ Simulate sending and receiving messages between processes. """
        for _ in range(5):
            # Perform an internal event
```

```python
            self.perform_event()
            time.sleep(random.random())  # Random sleep to simulate concurrency

            receiver = random.choice(peers)  # Randomly pick a process to send a message to
            if receiver != self:  # Don't send to itself
                self.send_message(receiver, f"Message from Process {self.pid}")
            time.sleep(random.random())  # Random sleep to simulate concurrency


# Simulating vector clock across multiple processes using threading
def start_simulation():
    num_processes = 3
    processes = [VectorClock(i, num_processes) for i in range(num_processes)]

    threads = []
    for process in processes:
        thread = threading.Thread(target=process.run, args=(processes,))
        threads.append(thread)
        thread.start()

    # Wait for all threads to complete
    for thread in threads:
        thread.join()

start_simulation()
```