1. RPC Servers

Client code

```
from xmlrpc.client import ServerProxy

server_proxy = ServerProxy("http://localhost:5000")

output = server_proxy.factorial(5)

print(f'Factorial of the given number is {output}')
```

Server Code:

```
from xmlrpc.server import SimpleXMLRPCServer

def factorial(num):
    res = 1
    if num == 0 and num == 1:
        return res
    else:
        for i in range(num, 1, -1):
            res *= i
        return res

server = SimpleXMLRPCServer(("localhost", 5000))
server.register_function(factorial)

print("Server is running on https://localhost:5000")
server.serve_forever()
```

2. Multithread application

```
import threading
import time
import os

project_group_member_roll_no_arr = ["BEA128", "BEA139", "BEA160", "BEA171"]
project_group_member_name_arr = ["Amogh", "Dhruv", "Shoaib", "Mudit"]

def project_group_member_roll_no():
    for roll_no in project_group_member_roll_no_arr:
        time.sleep(1)
        print(roll_no)
        print(f"Task {roll_no} assigned to thread: 
{format(threading.current_thread().name)}")
        print(f"ID of process running task {roll_no}: 
{format(os.getpid())}\n")

def project_group_member_name():
    for name in project_group_member_name_arr:
        time.sleep(1)
        print(name)
        print(f"Task {name} assigned to thread: 
{format(threading.current_thread().name)}")
        print(f"ID of process running task {name}: {format(os.getpid())}\n")

thread1 = threading.Thread(target=project_group_member_roll_no)
thread2 = threading.Thread(target=project_group_member_name)

thread1.start()
thread2.start()

thread1.join()
thread2.join()

print("Both threads have finished.")
```

4. Load Balancer

```
class RoundRobinLoadBalancer:
    def __init__(self, servers):
        self.servers = servers
        self.current_index = 0

    def get_next_server(self):
        server = self.servers[self.current_index]
        self.current_index = (self.current_index + 1) % len(self.servers)
        return server

servers_list = ['Server 1', 'Server 2', 'Server 3', 'Server 4', 'Server 5']
load_balancer = RoundRobinLoadBalancer(servers_list)

for i in range(17):
    next_server = load_balancer.get_next_server()
    print(f"Request {i + 1} routed to {next_server}")
```

or

```
from itertools import cycle

class LoadBalancer:
    def __init__(self, servers):
        self.servers = cycle(servers)
    def get_server(self):
        return next(self.servers)

if __name__ == '__main__':
    servers = ['server 1', 'server 2', 'server 3', 'server 4', 'server 5']
    lb = LoadBalancer(servers=servers)

    for job in range(17):
        server = lb.get_server()
        print(f'Request for job {job} is handled by {server}')
```

5. Lamport Clock

```python
from multiprocessing import Process, Pipe
from os import getpid
from datetime import datetime

def local_time(counter):
    return f'(LAMPORT TIME ={counter}, LOCAL TIME = {datetime.now()})'

def calc_recv_timestamp(recv_time_stamp, counter):
    return max(recv_time_stamp, counter) + 1

def event(pid, counter):
    counter += 1
    print('\nEvent happened in {} !'.format(pid) + local_time(counter))
    return counter

def send_message(pipe, pid, counter):
    counter += 1
    pipe.send(('Empty shell', counter))
    print('\nMessage sent from ' + str(pid) + local_time(counter))
    return counter

def recv_message(pipe, pid, counter):
    message, timestamp = pipe.recv()
    counter = calc_recv_timestamp(timestamp, counter)
    print('\nMessage received at ' + str(pid) + local_time(counter))
    return counter

def process_one(pipe12):
    pid = getpid()
    counter = 0
    counter = event(pid, counter)
    counter = send_message(pipe12, pid, counter)
    counter = event(pid, counter)
    counter = recv_message(pipe12, pid, counter)
    counter = event(pid, counter)

def process_two(pipe21, pipe23):
    pid = getpid()
    counter = 0
    counter = recv_message(pipe21, pid, counter)
    counter = send_message(pipe21, pid, counter)
    counter = send_message(pipe23, pid, counter)
    counter = recv_message(pipe23, pid, counter)

def process_three(pipe32, pipe34):
    pid = getpid()
    counter = 0
    counter = recv_message(pipe32, pid, counter)
    counter = send_message(pipe32, pid, counter)
    counter = send_message(pipe34, pid, counter)
    counter = recv_message(pipe34, pid, counter)

def process_four(pipe43, pipe45):
    pid = getpid()
    counter = 0
    counter = recv_message(pipe43, pid, counter)
    counter = send_message(pipe43, pid, counter)
    counter = send_message(pipe45, pid, counter)
    counter = recv_message(pipe45, pid, counter)

def process_five(pipe54):
    pid = getpid()
    counter = 0
    counter = recv_message(pipe54, pid, counter)
    counter = send_message(pipe54, pid, counter)

if __name__ == '__main__':
    oneTwo, twoOne = Pipe()
    twoThree, threeTwo = Pipe()
    threeFour, fourThree = Pipe()
    fourFive, fiveFour = Pipe()

    process1 = Process(target=process_one, args=(oneTwo,))
    process2 = Process(target=process_two, args=(twoOne, twoThree))
    process3 = Process(target=process_three, args=(threeTwo,threeFour))
    process4 = Process(target=process_four, args=(fourThree, fourFive))
    process5 = Process(target=process_five, args=(fiveFour, ))

    process1.start()
    process2.start()
    process3.start()
    process4.start()
    process5.start()

    process1.join()
    process2.join()
    process3.join()
    process4.join()
    process5.join()
```

6. Bully Algorithm

```
import random

class Node:
    def __init__(self, id):
        self.id = id
        self.alive = True
        self.coordinator = None

    def election(self, nodes):
        higher_nodes = [node for node in nodes if node.id > self.id and 
node.alive]
        if not higher_nodes:
            self.coordinator = self
            for node in nodes:
                if node != self:
                    node.notify_elected(self)
        else:
            higher_nodes.sort(key=lambda x: x.id)
            highest_node = higher_nodes[-1]
            highest_node.start_election(nodes)

    def start_election(self, nodes):
        print(f"Node {self.id} starts the election.")
        higher_nodes = [node for node in nodes if node.id > self.id and 
node.alive]
        if not higher_nodes:
            self.coordinator = self
            for node in nodes:
                if node != self:
                    node.notify_elected(self)
        else:
            for node in higher_nodes:
                node.election(nodes)

    def notify_elected(self, coordinator):
        print(f"Node {self.id} is notified of new coordinator: Node 
{coordinator.id}")
        self.coordinator = coordinator

    def crash(self):
        self.alive = False
        print(f"Node {self.id} crashed.")

    def __str__(self):
        return f"Node {self.id}, Coordinator: {self.coordinator.id if
self.coordinator else None}"


if __name__ == "__main__":
    num_nodes = 5
    nodes = [Node(i) for i in range(1, num_nodes + 1)]
    nodes[random.randint(1, 5)].start_election(nodes)
    nodes[random.randint(1, 5)].crash()
    print(nodes[2])
    print(nodes[3])
```

7. Mutual Exclusion Lamport

```python
from threading import Thread
import time

class LamportMutex:
    def __init__(self, num_processes):
        self.num_processes = num_processes
        self.clock = [0] * num_processes
        self.in_cs = [False] * num_processes
        self.queue = []

    def request_cs(self, pid):
        self.clock[pid] += 1
        self.queue.append((self.clock[pid], pid))
        self.queue.sort()
        print(f"Process {pid} is requesting to enter the critical section.")
        while self.queue[0][1] != pid or self.queue[0][0] != self.clock[pid]:
            time.sleep(0.1)
            if self.in_cs[self.queue[0][1]]:
                print(f"Process {pid} is waiting to enter the critical
section.")
        self.in_cs[pid] = True

    def release_cs(self, pid):
        self.in_cs[pid] = False
        self.queue.pop(0)

def process(mutex, pid):
    while True:
        mutex.request_cs(pid)
        print(f"Process {pid} is in the critical section.")
        time.sleep(1)
        print(f"Process {pid} is exiting the critical section.")
        mutex.release_cs(pid)
        time.sleep(1)

if __name__ == "__main__":
    num_processes = 3
    mutex = LamportMutex(num_processes)
    threads = []

    for i in range(num_processes):
        t = Thread(target=process, args=(mutex, i))
        threads.append(t)
        t.start()

    for t in threads:
        t.join()
```

8. Deadlock management

```python
import threading
import time

lock1 = threading.Lock()
lock2 = threading.Lock()

def deadlock_thread1():
    if lock1.acquire(timeout=2):
        print("Thread 1 acquired lock 1")
        time.sleep(1)
        print("Thread 1 waiting to acquire lock 2")
        if lock2.acquire(timeout=2):
            print("Thread 1 acquired lock 2")
            lock2.release()
        else:
            print("Thread 1 failed to acquire lock 2, terminating...")
        lock1.release()
    else:
        print("Thread 1 failed to acquire lock 1, terminating...")

def deadlock_thread2():
    if lock2.acquire(timeout=2):
        print("Thread 2 acquired lock 2")
        time.sleep(1)
        print("Thread 2 waiting to acquire lock 1")
        if lock1.acquire(timeout=2):
            print("Thread 2 acquired lock 1")
            lock1.release()
        else:
            print("Thread 2 failed to acquire lock 1, terminating...")
        lock2.release()
    else:
        print("Thread 2 failed to acquire lock 2, terminating...")

thread1 = threading.Thread(target=deadlock_thread1)
thread2 = threading.Thread(target=deadlock_thread2)

thread1.start()
thread2.start()

thread1.join()
thread2.join()

print("Program finished execution")
```

9. Name Resolution Protocol

```
import socket


def get_ip_address(url):
    try:
        host_name = socket.gethostbyname(url)
        host_ip = socket.gethostbyname(host_name)
        print("Hostname:", host_name)
        print("IP:", host_ip)
    except:
        print("Unable to get hostname and IP")


if __name__ == '__main__':
    url = "www.ltce.in"
    get_ip_address(url)
```

10. Distributed Shared Memory

```
import threading

memory = {}

lock = threading.Lock()

def set_value(key, value):
    global memory
    global lock
    lock.acquire()
    memory[key] = value
    lock.release()

def get_value(key):
    global memory
    global lock
    lock.acquire()
    value = memory.get(key, None)
    lock.release()
    return value

def thread_1():
    set_value("a", 25)
    set_value("b", 18)
    print("Thread 1 sets value of a as 25 and b as 18")

def thread_2():
    value_a = get_value("a")
    value_b = get_value("b")
    print("Thread 2 reads value of a as {} and b as {}".format(value_a,
value_b))
    set_value("c", value_a * value_b)

def main_thread():
    thread1 = threading.Thread(target=thread_1)
    thread2 = threading.Thread(target=thread_2)
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()
    value_c = get_value("c")
    print("Main thread reads value of c as {}".format(value_c))

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