

**Assignment****1****Telecom Network Dataset:**

Node	Latency (ms)	Throughput (Mbps)	PacketLoss (%)	CPU (%)	Memory (GB)	Events
Edge1	12	500	0.2	40	4	RPCCall, TransactionCommit
Edge2	15	480	0.5	45	4.5	RPCCall, NodeFailure
Core1	8	1000	0.1	60	8	TransactionCommit
Core2	10	950	0.2	55	7.5	NodeFailure, Recovery
Cloud1	20	1200	0.3	70	16	RPCCall, NodeFailure

- (a) Engineer distributed system architecture connecting all nodes.
  - (b) Write Java code modeling nodes, events, message-passing methods.
  - (c) Identify, prioritize performance bottlenecks; justify mathematically.
- (a) Engineer distributed system architecture connecting all nodes.

## **Proposed Architecture: Hierarchical Edge–Core–Cloud Distributed Architecture**

### **1. Architecture Layers**

#### **1. Edge Layer (Edge1, Edge2)**

- Close to users
- Handles:
  - RPC calls
  - Initial request validation
  - Lightweight transactions
- Characteristics:
  - Moderate latency (12–15 ms)
  - Lower throughput (480–500 Mbps)
  - Limited memory (4–4.5 GB)

#### **2. Core Layer (Core1, Core2)**

- Backbone of the system
- Handles:

- Transaction Commit
- Failure recovery
- Coordination
- Characteristics:
  - Lowest latency (8–10 ms)
  - High throughput (~950–1000 Mbps)
  - Moderate CPU usage

### 3. Cloud Layer (Cloud1)

- Centralized analytics & heavy processing
- Handles:
  - Large RPC calls
  - Long-term storage
  - Global coordination
- Characteristics:
  - Highest latency (20 ms)
  - Highest throughput (1200 Mbps)
  - Large memory (16 GB)

## 2. Communication Model

- **RPC-based communication** between nodes
- **Asynchronous messaging** (Kafka / RabbitMQ conceptually)
- **Failure detection** using heartbeat messages
- **Consensus handled at Core layer**

### 3. Data Flow

Client → Edge → Core → Cloud(Request)

Cloud → Core → Edge (response)

## 4. Fault Tolerance

- NodeFailure handled via:
  - Replication
  - Failover at Core nodes

- Recovery events handled by Core2

(b) Write Java code modeling nodes, events, message-passing methods.

```

import java.util.*;

class Node {
    String name;
    int latency;
    int throughput;
    double packetLoss;
    int cpu;
    double memory;
    List<String> events;

    public Node(String name, int latency, int throughput, double packetLoss,
                int cpu, double memory, List<String> events) {
        this.name = name;
        this.latency = latency;
        this.throughput = throughput;
        this.packetLoss = packetLoss;
        this.cpu = cpu;
        this.memory = memory;
        this.events = events;
    }

    public void sendMessage(Node receiver, String message) {
        System.out.println(this.name + "Sending message to:" + receiver.name);
        simulateLatency();
        receiver.receiveMessage(message);
    }

    private void simulateLatency() {
        try {
            Thread.sleep(latency);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void receiveMessage(String message) {
        System.out.println(this.name + "received message: " + message);
    }
}

```

```
public class DistributedSystem {
```

```
    public static void main(String[] args) {
```

```
        Node edge1 = new Node("Edge1", 12, 500, 0.2, 40, 4);
```

```
        Arrays.asList("RPCcall", "TransactionCommit");
```

```
        Node core1 = new Node("Core1", 8, 1000, 0.1, 60, 8);
```

```
        Arrays.asList("TransactionCommit");
```

```
        Node cloud1 = new Node("cloud1", 20, 1200, 0.3, 70, 16);
```

```
        Arrays.asList("RPCcall", "NodeFailure");
```

```
        edge1.sendMessage(core1, "User Request");
```

```
        core1.sendMessage(cloud1, "Process Transaction");
```

```
    }
```

```
}
```

(c) Identify, prioritize performance bottlenecks; justify mathematically.

### Key Metrics

- Latency
- Throughput
- Packet Loss
- CPU utilization

## 1. Latency Bottleneck

Cloud1 has **20 ms**, highest latency.

**End-to-End Latency:**

$$L_{total} = L_{edge} + L_{core} + L_{cloud} \quad L_{total} = L_{edge} + L_{core} + L_{cloud} = 12 + 8 + 20 = 40 \text{ ms}$$

Cloud contributes **50%** of total latency.

## 2. Throughput Bottleneck

Edge nodes:

- Edge1 = 500 Mbps
- Edge2 = 480 Mbps

**System throughput limited by minimum throughput:**

$$T_{system} = \min(500, 480, 1000, 1200) = 480 \text{ Mbps} \quad T_{system} = \min(500, 480, 1000, 1200) = 480 \text{ Mbps}$$

→ **Edge2 is a throughput bottleneck**

## 3. CPU Bottleneck

Cloud1 CPU = **70%**

High CPU increases queueing delay:

$$W \propto \frac{1}{1 - \rho} \quad W \propto \frac{1}{1 - \rho}$$

Where  $\rho$  = CPU utilization  $\rho$  = CPU utilization

At 70%, waiting time increases rapidly.

## Telecom Network Dataset:

Node	Latency (ms)	Throughput (Mbps)	PacketLoss (%)	CPU (%)	Memory (GB)	Services
EdgeA	10	520	0.3	40	4	RPCall, DataReplication
EdgeB	14	470	0.5	48	5	RPCall, Migration
CoreX	7	980	0.1	65	8	TransactionCommit
CoreY	9	950	0.2	58	7	Recovery, LoadBalancing
CloudZ	22	1250	0.4	72	16	Analytics, RPCall

- Identify node causing maximum latency; justify using dataset.
- Allocate processes to balance CPU and memory load across nodes.
- Write Python code simulating process allocation and load balancing.

(a) Identify node causing maximum latency; justify using dataset.

From our dataset

Node	Latency (ms)
EdgeA	10
EdgeB	14
CoreX	7
CoreY	9
CloudZ	<b>22</b>

## Answer

**CloudZ has the highest Latency value hence causing the Maximum Latency**

(b) Allocate processes to balance CPU and memory load across nodes.

## Goal

Balance:

- CPU load
- Memory usage

Service	Node	Reason
RPCCall	EdgeA	Low latency
Migration	EdgeB	Medium CPU
TransactionCommit	CoreX	Low latency, high CPU
LoadBalancing	CoreY	Designed for balancing
Analytics	CloudZ	High memory (16GB)

(c) Write Python code simulating process allocation and load balancing.

```
class Node:
```

```
    def __init__(self, name, cpu, memory):
```

```
        self.name = name
```

```
        self.cpu = cpu
```

```
        self.memory = memory
```

```
        self.processes = []
```

```
    def assign(self, process, cpu_cost, mem_cost):
```

```
        if self.cpu + cpu_cost <= 80 and self.memory >= mem_cost:
```

```
            self.cpu += cpu_cost
```

```
            self.memory -= mem_cost
```

```
            self.processes.append(process)
```

```
nodes = [
```

```
    Node("Edge A", 40, 4),
```

```
    Node("Edge B", 48, 5),
```

```
    Node("Core X", 65, 8),
```

```
    Node("Core Y", 58, 7),
```

```
    Node("Cloud Z", 72, 16),
```

```
]
```

```
processes = [
```

```
    ("Analytics", 5, 4),
```

```
    ("TransactionCommit", 4, 2),
```

```
    ("RPCcall", 3, 1)
```

```
]
```

```
for process in processes:
```

```
    for node in nodes:
```

```
        node.assign(*process)
```

```
for node in nodes:
```

```
    print(node.name, node.processes)
```

## Telecom Transaction &amp; Concurrency Dataset:

Node	CPU (%)	Memory (GB)	Latency (ms)	Transactions/sec	Locks (%)	Services
Edge1	45	4	12	120	5	RPC, EventOrdering
Edge2	50	4.5	15	100	8	RPC, NodeFailureRecovery
Core1	60	8	8	250	12	2PC/3PC, TransactionCommit
Core2	55	7.5	10	230	10	DeadlockDetection, LoadBalancing
Cloud1	70	16	20	300	15	DistributedSharedMemory, Analytics

a) Identify nodes causing transaction bottlenecks.

### Indicators

- High locks
- High CPU
- High latency

Node	Locks (%)	Transactions/sec
Core1	12	250
Core2	10	230
Cloud1	15	300

### Answer

**Core1 & Cloud1** are transaction bottlenecks

(b) Engineer consensus or commit protocol to maximize throughput.

### Chosen Protocol: Optimized 2PC with Fallback to 3PC

Why?

- Core nodes have low latency
- Cloud node handles failures

## Flow

1. Prepare phase (Core nodes)
2. Commit phase
3. Timeout → switch to 3PC

(c) Write Java code to implement a deadlock resolution strategy.

Uses **Wait-Die / Wound-Wait concept**

```

3 import java.util.*;

class Transaction {
    int id;
    int timestamp;

    Transaction (int id, int timestamp) {
        this.id = id;
        this.timestamp = timestamp;
    }
}

public class DeadlockResolver {
    public static void resolve (List<Transaction> waiting) {
        Transaction youngest = waiting.stream()
            .max (Comparator.comparingInt (t -> t.timestamp))
            .get ();
        System.out.println ("Aborting Transaction" + youngest.id);
    }
}

```

## Telecom Fault Tolerance &amp; Capstone Dataset:

Node	CPU (%)	Memory (GB)	Latency (ms)	Throughput (Mbps)	FailureType	Services
Edge1	45	4	12	500	Crash	RPCall, DataReplication
Edge2	50	4.5	15	470	Omission	RPCall, Migration
Core1	65	8	8	980	Byzantine	TransactionCommit
Core2	58	7	10	950	Crash	Recovery, LoadBalancing
Cloud1	72	16	22	1250	Omission	Analytics, RPCall

- (a) Identify nodes most likely to cause system failure.
- (b) Write Python code implementing a redundancy and failover strategy.
- (c) Engineer replication and access control for distributed file systems.
- (d) Integrate edge-core-cloud services for an end-to-end carrier-grade system.

(a) Identify nodes most likely to cause system failure.

**Node      Failure Type**

Core1      **Byzantine**

Cloud1    Omission

Edge2     Omission

**Answer**

**Core1 is most dangerous (Byzantine failures)**

(b) Write Python code implementing a redundancy and failover strategy.

4b

```
def failover(primary, backup, status):  
    if status == "FAILED":  
        print(f"{primary} failed. Switching to {backup}")  
    else:  
        print(f"{primary} operational")  
failover("core1", "Core 2", "FAILED")
```

(c) Engineer replication and access control for distributed file systems.

## **Replication**

- Primary–Secondary replication
- Quorum-based reads/writes

## **Access Control**

- Role-Based Access Control (RBAC)
- Edge: Read
- Core: Read/Write
- Cloud: Admin

(d) Integrate edge-core-cloud services for an end-to-end carrier-grade system.

- Edge handles real-time RPC
- Core handles coordination & recovery
- Cloud handles analytics & storage
- Redundancy + Replication
- Consensus + Failover