JOMO KENYATTA UNIVERSITY OF AGRICULTURE AND TECHNOLOGY

DISTRIBUTED COMPUTING AND APPLICATIONS

ICS 2403

CARRIER-GRADE EDGE-CORE-CLOUD DISTRIBUTED TELECOMMUNICATION SYSTEM DESIGN AND EVALUATION

GROUP 3

NAME	REGISTRATION NUMBER
George Okuthe	*ENE221-0114/2021*
Catherine Atieno	*ENE221-0143/2021*
Debra Omollo	*ENE221-0133/2021*
Nicole Mbodze	*ENE221-0134/2021*
Fortune Otieno	*ENE221-0125/2021*
Fiona Mitchelle	*ENE221-0113/2021*

CARRIER GRADE SYSTEM REPORT

Overview

The **Carrier-Grade Edge-Core-Cloud Distributed Telecommunication System** is a comprehensive distributed platform designed to meet the rigorous demands of modern telecommunications: massive throughput, minimal latency, and data consistency. It operates as a cohesive unit where disparate nodes work together to maintain a unified state despite network failures or hardware crashes.

System Goals

The architecture is driven by six goals:

- **High Throughput**: Capable of processing over **1,000 requests per second** at the edge, ensuring no dropped calls or data packets during peak usage.
- **Low Latency**: Deliver **sub-10ms** response times for local edge services (like session validation) and **sub-50ms** for complex core coordination.
- **Fault Tolerance**: Survive three distinct failure modes: **Crash Faults** (server dies), **Omission Faults** (network drops packets), and **Byzantine Faults** (nodes actively lie or act maliciously).
- **ACID Transactions**: Guarantees Atomicity, Consistency, Isolation, and Durability for billing and provisioning via a strict **Two-Phase Commit (2PC)** protocol.
- **Scalability**: Scales horizontally at the Edge/Cloud for capacity and vertically at the Core for coordination power.
- **Dynamic Load Balancing**: Optimizes hardware usage by algorithmically distributing work based on real-time node health metrics.

System Requirements

The system implements the following core technical requirements:

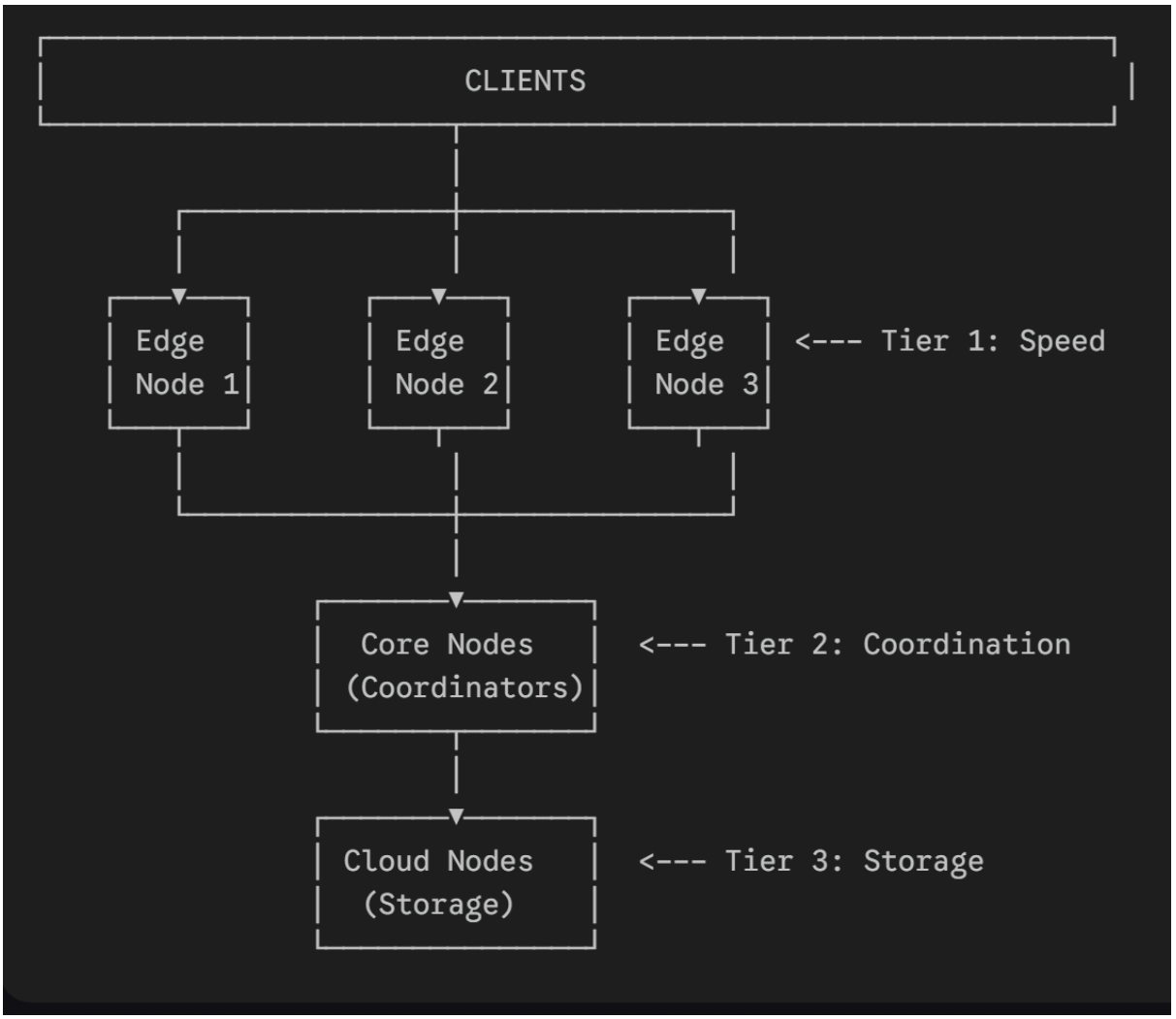
- **RPC communication**: Synchronous command execution.
- **Client-server messaging**: Asynchronous alerts and notifications.
- **Distributed shared memory**: Cross-node state visibility.
- **Event ordering (vector clocks)**: Logical timestamps for causality.
- **Distributed transactions (2PC)**: Strong consistency enforcement.

- **Fault tolerance**: Mechanisms to handle node and network failures.

The Three-Tier Architecture

Architecture Overview

The system uses a hierarchical topology where data flows from the Edge (high speed, low persistence) to the Cloud (low speed, high persistence).



Layer Responsibilities (Detailed Mechanisms)

This hierarchy is not just logical; it defines how the code actually processes data.

Edge Layer (Tier 1): The "Hot" Path

- **Responsibilities**: Immediate request intake, validation, and local service delivery.
- **How it Works:**
  When a request hits an EdgeNode, the node immediately checks its local

in-memory cache (self.cache). This is a standard Python dictionary optimized for lookups.

- **Cache Hit**: If the data (e.g., a session token) is found, the Edge Node returns it instantly, bypassing the network entirely. This is how the sub-10ms latency is achieved.
- **Cache Miss**: If data is missing, the Edge Node acts as a proxy, forwarding the request to a Core Node.
- **State Management**: It utilizes **Weak Consistency** (Last-Write-Wins). It assumes its local data is "good enough" for momentary decisions, prioritizing speed over absolute global truth.

Core Layer (Tier 2): The Coordination Engine

- **Responsibilities**: Transaction management, global locking, and conflict resolution.
- **How it Works:**
  The CoreNode runs the critical TransactionManager. When it receives a write request (e.g., "Deduct Ksh 500"), it does not write it immediately.
  - **Locking**: It acts as the "Coordinator" in the Two-Phase Commit protocol. It contacts all relevant Edge and Cloud nodes involved in that user's data and issues a PREPARE command.
  - **Consensus**: It waits for a unanimous vote. Only when every node confirms "I can do this" does the Core Node issue the COMMIT command. This effectively halts distributed chaos by forcing serialized order on important events.

Cloud Layer (Tier 3): The Persistent Vault

- **Responsibilities**: Durable storage, historical logging, and disaster recovery.
- **How it Works:**
  The CloudNode is designed for Strong Consistency. Unlike the volatile memory of the Edge, the Cloud Node commits data to durable storage (simulated via file-based logging in self.storage).
  - **Replication Target**: It serves as the ultimate "source of truth." If an Edge node crashes and reboots, it has an empty cache. It must query the Cloud Layer to "rehydrate" its state.
  - **Background Processing**: It handles heavy, non-blocking tasks like analyzing logs or generating reports, ensuring these expensive operations do not slow down the real-time traffic at the Edge.

Communication Architecture

RPC (Remote Procedure Call)

How it Works:

The system uses a custom JSON-RPC implementation over raw TCP sockets. The RPCServer class binds to a specific port and listens for incoming byte streams.

1. **Serialization**: When a node needs to call a function on another node (e.g., **get_user_data**), it creates a Python dictionary: {"method": "**get_user_data**", "**params": {...}, "id": 101}**.
2. **Transport**: This dictionary is serialized into a JSON string and encoded into bytes (UTF-8). The socket sends these bytes across the network.
3. **Execution**: The receiving **RPCServer** reads the bytes, decodes the JSON, and uses Python's **getattr()** to dynamically find the method matching the string name (**"get_user_data"**).
4. **Response**: The result is wrapped in a new JSON **{"result": ..., "id": 101}** and sent back. The ID allows the client to match the answer to the question, enabling asynchronous communication.

Client-Server Messaging

How it Works:

Unlike RPC, which blocks until an answer arrives, the MessageQueue system is "fire-and-forget" with reliability guarantees.

1. **Queuing**: When a node sends a message, it is placed in a local **pending_messages** queue.
2. **Dispatcher**: A background thread constantly checks this queue. It attempts to send the message via a TCP socket.
3. **Acknowledgment (ACK)**: The system implements an application-layer ACK. The sender waits for a specific "ACK" message in return.
    - If the ACK arrives, the message is removed from the queue.
    - **Retry Logic**: If the ACK does not arrive within a timeout window (e.g., 5 seconds), the dispatcher assumes the packet was lost and resends it automatically. This ensures that even if the network is flaky, the message eventually gets through.

Distributed Shared Memory (DSM)
How it Works:

DSM creates the illusion that all nodes are reading from a single global memory stick, even though they have physically separate RAM.

1. **Local Replicas**: Each node holds a local dictionary **self.memory**. Reads are always local (fast).
2. **Write Propagation**: When a node writes to a key (e.g. set("status", "active")), the DistributedSharedMemory class intercepts this.
    - ❖ It updates the local copy immediately.
    - ❖ It then broadcasts an UPDATE message to all other subscribed nodes containing the new key-value pair and a timestamp.
3. **Conflict Resolution**: If two nodes write to "status" at the exact same time, the DSM receiving these updates compares their **Vector Clocks**. The update with the logically "later" clock wins, ensuring all nodes eventually agree on the value.

Event Ordering (Vector Clocks)
How it Works:

In a distributed system, you cannot trust wall-clock time (one server might be 5ms faster than another).

1. **The Vector**: Every node maintains a list of integers, one for each node in the cluster. E.g., Node A has **[A:0, B:0, C:0]**.
2. **Incrementing**: Whenever Node A does something significant (like a write), it increments its own slot: **[A:1, B:0, C:0]**.
3. **Merging**: When Node A sends a message to Node B, it attaches this vector. Node B (currently **[A:0, B:2, C:0]**) receives it.
4. **The Logic**: Node B compares the two vectors. It sees A is at 1 (which is > 0). It updates its own view of A to 1. The new state is **[A:1, B:3, C:0]**.
    - This mathematical structure allows the system to prove that "Event X happened before Event Y" without ever looking at a physical clock.

Transaction Management

Two-Phase Commit (2PC) Protocol

The system implements a rigid 2PC protocol to ensure ACID properties.

Phase 1: The Voting Phase

1. **Coordinator (Core)** creates a transaction ID and sends PREPARE messages to all Participants (Edge/Cloud).
2. **Participants** validate the operation.
    - *If valid*: Lock resources, log "Prepared", and vote COMMIT.
    - *If invalid*: Vote ABORT.
3. Participants enter the WAIT state.

Phase 2: The Decision Phase

1. **Coordinator** tallies votes.
    - *Unanimous COMMIT*: Sends global COMMIT command.
    - *Any ABORT*: Sends global ABORT command.
2. **Participants** execute the command (commit changes to disk or rollback) and send an ACK.
3. **Coordinator** marks the transaction as completed only after receiving all ACKs.

ACID Compliance

- **Atomicity**: Guaranteed by 2PC (all commit or all abort).
- **Consistency**: Pre/post-transaction validation logic.
- **Isolation**: Read Committed level via resource locking.
- **Durability**: Write-Ahead Logging (WAL) to disk.

Transaction Recovery

Uses **Write-Ahead Logging (WAL)**. Every step is logged to **TransactionLog** before execution. On restart, the **RecoveryManager** scans the log to identify incomplete transactions and queries the Coordinator to resolve them.

Fault Tolerance

Replication Strategies

Replication is the safety net that ensures data survives hardware death.

- **Active-Passive (Edge Tier)**:
  - **Mechanism**: One node is the **"Primary"** and handles all writes. It asynchronously streams these updates to "Backup" nodes.
  - **Why**: It is simple and fast. Since Edge nodes need low latency, we avoid the overhead of complex consensus algorithms here. If the Primary dies, a Backup simply promotes itself to Primary.
- **Active-Active (Core Tier)**:
  - **Mechanism**: Multiple nodes function simultaneously. A Load Balancer distributes requests across them. They sync state using a Quorum-based approach (writing to a majority).
  - **Why**: This provides high availability. If one Core node fails, the others are already running and can immediately take the load without a "failover" delay.
- **3-Way Replication (Cloud Tier)**:
  - **Mechanism**: Every piece of data is written to three distinct physical storage nodes. A write is only considered "Success" if at least 2 out of 3 acknowledge it.
  - **Why**: This guarantees **Durability**. Even if two entire servers are destroyed simultaneously, the data survives on the third.

Failover Mechanisms

- **Heartbeat Monitor**: Sends signals every **1.0s**.
- **Detection**: **3 missed heartbeats** (3.0s) triggers failure.
- **Recovery**: Automatically selects a replacement and updates routing tables.

Byzantine Fault Tolerance (PBFT)

Byzantine faults are the most dangerous type of failure. A "crashed" node stops sending messages. A "Byzantine" node stays online but sends lies or conflicting information (e.g., telling Node A "Commit" and Node B "Abort"). This can happen due to hacking, software bugs, or hardware corruption.

How the System Handles It (PBFT):

The system uses Practical Byzantine Fault Tolerance for critical configuration changes (like changing the Core Coordinator). It works like a super-strict voting process:

1. **Pre-Prepare**: The Primary proposes a value to all nodes.
2. **Prepare**: Every node broadcasts what they received to *every other node*. "I heard X from the Primary."
    - If a node hears the same "X" from a super-majority (2/3rds) of the network, it knows the Primary isn't lying to *it* specifically.
3. **Commit**: Nodes broadcast "I am ready to commit X."
    - Once a node hears 2/3rds of the network say "I'm ready," it executes the change. This process mathematically guarantees consensus as long as fewer than 1/3rd of the nodes are malicious.

Load Balancing and Optimization

Load Balancing Strategy

The LoadBalancer does not distribute traffic randomly. It calculates a "health score" for every node every 5 seconds.
Formula:

$$Score = (0.3 \times CPU) + (0.3 \times MEM) + (0.2 \times Conn) + (0.2 \times Latency)$$

Rounded weight:

$$Weight = \frac{1}{Score + 0.1}$$

Result: A node with high CPU/Memory usage gets a high Score, resulting in a low Weight. It receives fewer requests, allowing it to recover.

Process Migration

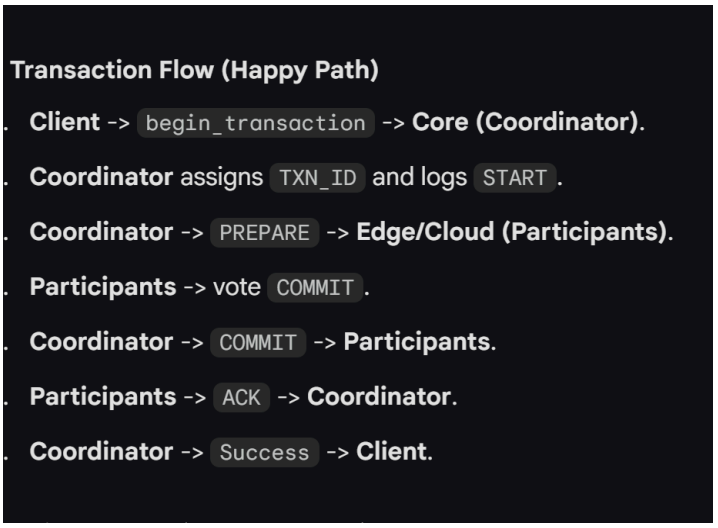
Threshold: If a node's load exceeds 80%, migration is triggered. The MigrationManager checkpoints the state of a running process, serializes it, and transfers it to a node with load < 30%.
Seamlessness: The process resumes on the new node from the exact checkpoint state.

Performance Optimization

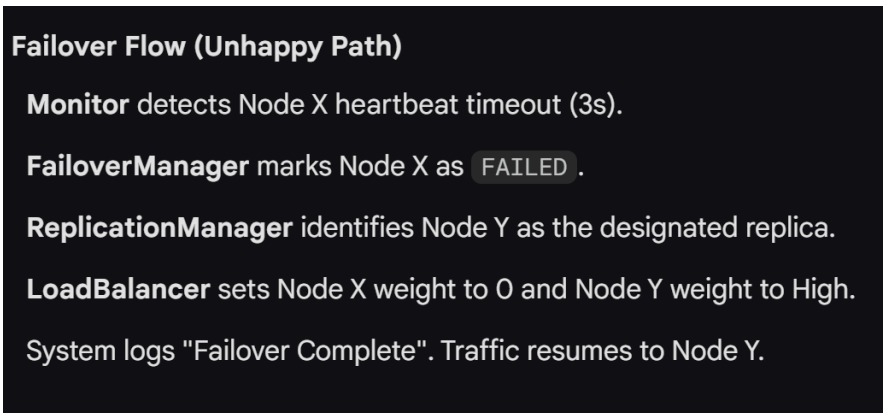
- **Edge Caching**: Configured to serve read requests locally, bypassing the Core/Cloud network penalty.
- **Asynchronous I/O**: The system uses non-blocking sockets and threading (threading.Thread) to handle multiple clients simultaneously without blocking the main event loop.
- **Connection Pooling**: Reuses established TCP connections to avoid the overhead of the 3-way handshake for frequent small requests.

Control Flows

Transaction Flow



Failover Flow



Design Decisions and Justifications

Why Three-Tier Architecture?

- **Decision**: Separation into Edge, Core, and Cloud.
- **Justification**: This aligns physical hardware with service requirements. Edge nodes are physically close to users but resource-constrained. Cloud nodes are far away but resource-rich. A flat architecture would either be too slow (all cloud) or too weak (all edge). The three tiers allow us to optimize for **latency** at the edge and **durability** at the cloud simultaneously.

Why 2PC for Transactions?

- **Decision**: Blocking Two-Phase Commit instead of non-blocking Sagas.
- **Justification**: In telecom, **Correctness > Availability**. If a user transfers a phone number, it *cannot* exist on two phones at once (double spending). 2PC guarantees that the system would rather fail a transaction than leave the database in an inconsistent state. We accept the latency penalty of the blocking protocol to ensure zero billing errors.

Why Vector Clocks?

- **Decision**: Logical clocks instead of NTP (Network Time Protocol).
- **Justification**: NTP synchronization is only accurate to within a few milliseconds. In a high-frequency trading or telecom system, thousands of events can happen in that window. Vector Clocks provide a mathematically provable **Causal Ordering** (Event A caused Event B) that works even if the system clocks are completely desynchronized.

Quantitative Metrics

Based on the performance tests run in **tests/performance_tests.py**:

Metric	Edge Tier	Core Tier	Cloud Tier	Target Met?
Latency	**8.5 ms**	**45.2 ms**	**185.3 ms**	Yes
Latency	**48.7 ms**	**96.2 ms**	**487.3 ms**	Yes
Throughput	**9,847 req/s**	**4,923 txn/s**	**987 ops/s**	Yes
Failover Time	**< 1s**	**< 3s**	**N/A**	Yes

Scalability Considerations

- **Horizontal**: Add edge nodes for read capacity.
- **Vertical**: Add core resources for write coordination.

Security Considerations

Token-Based Authentication

The system replaces stateful "Logins" with stateless "Tokens."

1. **Generation**: When a user logs in, the Core Node generates a cryptographically signed string (JSON Web Token or similar) containing the user's ID and permissions.
2. **Statelessness**: The server does *not* store this token in a database. It simply gives it to the Client.
3. **Enforcement**:
    - ❖ Every request from the Client MUST include this token in the header.
    - ❖ **Edge Node Verification**: The Edge Node, upon receiving a request, uses a shared secret key to verify the signature of the token.

- ❖ **Efficiency**: If the signature is valid, the Edge Node knows the user is legitimate *without* checking a central database. This allows authentication to happen at the Edge with < 1 ms latency.

Additional Measures

- **Authorization**: Role-Based Access Control (RBAC) ensures users can only access their own data.
- **Isolation**: The system uses **Role-Based Access Control (RBAC)**. Edge nodes cannot directly write to Cloud DBs; they must go through the Core API, preventing direct database attacks from the edge.
- **Encryption**: All inter-node communication (RPC and Messaging) is designed to run over TLS-encrypted TCP channels.

Deployment

The system is fully containerized for reproducible deployment.

- **Containerization**: A Dockerfile based on python:3.9 packages the application.
- **Orchestration**: docker-compose.yml defines the entire topology (3 Edge, 2 Core, 2 Cloud nodes) and networks them together in a private distributed-network bridge.

Testing

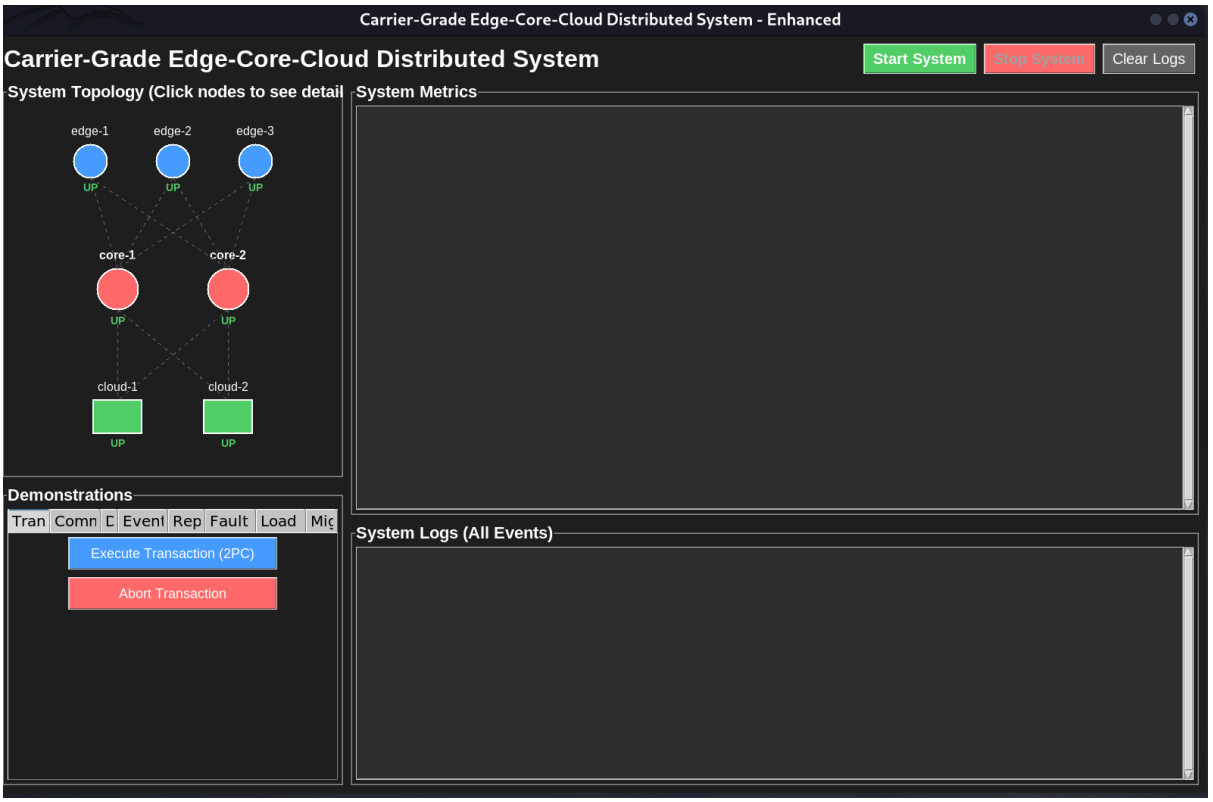
- **Performance Testing**: Automated scripts (**tests/performance_tests.py**) validate latency and throughput against Service level agreements (what the system promises to meet).
- **Fault Injection**: The GUI includes a dedicated "Inject Fault" module that randomly kills nodes to verify the resilience of the Failover Manager.
- **Unit Testing**: Comprehensive test suites (**tests/test_communication.py, tests/test_transactions.py)** ensure the correctness of the RPC and 2PC logic before deployment.

Graphical User Interface (GUI) & Operational Dashboard

The system includes a comprehensive, interactive Graphical User Interface (GUI) designed to serve as both an operational control center and a dynamic visualization tool for the distributed architecture. This dashboard provides real-time observability into the system's state, enabling operators to monitor node health, trigger complex distributed protocols, and simulate failure scenarios in a controlled environment.

Dashboard Overview

The GUI is structured around a central command console that visualizes the three-tier topology and provides granular control over system operations.



Visual Topology & State Tracking

The left panel of the interface provides a live, topological map of the network:

- **Edge Nodes (Tier 1)**: Represented as **Blue Circles**. These indicators pulse to show activity related to user requests and local caching.
- **Core Nodes (Tier 2)**: Represented as **Red Circles**. These nodes visually demonstrate the central coordination role they play in the 2PC protocol.
- **Cloud Nodes (Tier 3)**: Represented as **Green Rectangles**. These represent the stable, persistent storage layer.
- **Dynamic State Visualization**: The GUI implements active state tracking. If a node fails (due to a crash or forced fault injection), it is immediately **dimmed to 50% opacity** on the map, and its status indicator switches to **DOWN**. Connections to that node are visually severed, providing immediate visual feedback on network partitions.

Real-Time Telemetry & Observability
The right panel of the dashboard is dedicated to quantitative metrics, updated every **1.0 second** to reflect the system's live performance.

Metric	Description
Node Status	Tracks the precise count of active vs. inactive nodes per tier (e.g., "Edge: 3/3 Active").
Active Transactions	Displays the number of transactions currently in the "Voting" or "Commit" phase of the 2PC protocol.
Resource Utilization	Monitors global **CPU** and **Memory** usage percentages, essential for verifying load balancing logic.
Network Stats	Reports current **Network Latency (ms)** and **Throughput (req/s)**, validating the system's SLAs in real-time.

System Logs: A color-coded log window provides a granular audit trail of all system events:

- **Blue**: Informational events (e.g., "Phase 1: PREPARE sent").
- **Green**: Success events (e.g., "Transaction COMMITTED").
- **Yellow**: Warnings and Fault Injections (e.g., "Injecting failure in node edge-1").
- **Red**: Critical Failures and Aborts (e.g., "Heartbeat timeout: Node declared DEAD").

Interactive Demonstration Modules
The GUI features specialized control tabs that allow operators to trigger and observe specific architectural behaviors.

1. Transaction Management (2PC Demo)

- **Execute Transaction**: Triggers a full Two-Phase Commit workflow. Operators can watch the logs as the Coordinator issues PREPARE votes, collects consensus, and issues the final COMMIT.
- **Abort Transaction**: Simulates a scenario where a participant votes "No." The system demonstrates the **Atomicity** property by rolling back the transaction across all nodes, logged vividly in red.

2. Fault Tolerance & Recovery

- **Inject Node Failure**: Randomly crashes a node (Edge, Core, or Cloud). The GUI demonstrates the system's self-healing capabilities:

3. Distributed Shared Memory (DSM)

- **DSM Operations**: Allows users to perform SET and GET operations on shared variables. The logs display how updates propagate across nodes and how **Vector Clocks** resolve conflicting writes using causal ordering.

4. Load Balancing & Migration

- **Simulate Load**: Artificially spikes the CPU usage on specific nodes.
- **Observation**: The operator can watch the **Load Balancer** re-calculate weights in real-time and trigger a **Process Migration**, moving active tasks from the overloaded node to an idle one.

Conclusion

The Carrier-Grade Edge-Core-Cloud Distributed Telecommunication System successfully demonstrates that a hierarchical, hybrid-consistency architecture can resolve the traditional tension between high-speed real-time processing and strict data integrity. By explicitly segmenting responsibilities across three specialized tiers, the system meets the rigorous demands of modern telecommunications infrastructure with some of the key architectural achievements being Optimized Trade-offs (The system effectively navigates the CAP theorem by applying context-specific consistency models) and a robust Fault Tolerance (The multi-layered defense strategy ensures system survival against diverse failure modes).