Implementation Guide: Epidemic Replication

Distributed Systems - Exercise 4

Generated Guide

December 24, 2025

1 Overview

This document outlines the architecture and implementation steps for the distributed epidemic replication system. The system requires implementing a multi-versioned data architecture across three layers using Python[cite: 12, 65].

2 System Topology & Requirements

The system consists of 7 nodes divided into three layers, each with distinct consistency guarantees and replication strategies[cite: 23].

2.1 1. Core Layer (Strong Consistency)

- **Nodes:** A1, A2, A3 (Fully connected triangle) [cite: 46–49].
- **Data Version:** Holds the most modern versions $V = n$[cite: 20].
- **Strategy:** Update Everywhere, Active, Eager Replication[cite: 33].
- **Behavior:** When a node receives an update, it must broadcast it to all other core nodes and wait for acknowledgement before confirming the write.

2.2 2. Layer 1 (Causal Consistency)

- **Nodes:** B1, B2[cite: 50, 51].
- **Topology:** A2 connects to B1; A3 connects to B2[cite: 26].
- **Strategy:** Passive, Primary Backup[cite: 34].
- **Trigger:** Lazy propagation. Updates are sent every **10 updates**[cite: 34].

2.3 3. Layer 2 (Weak Consistency)

- **Nodes:** C1, C2[cite: 53, 55].
- **Topology:** B2 connects to both C1 and C2[cite: 26].
- **Strategy:** Passive, Primary Backup[cite: 35].
- **Trigger:** Lazy propagation. Updates are sent every **10 seconds**[cite: 35].

3 Implementation Steps

3.1 Step 1: Protocol & Node Structure

Communication must use Sockets or gRPC[cite: 32]. A generic `Node` class is recommended.

```python
import socket
import threading
import time
import json

class Node:
    def __init__(self, node_id, port, neighbors, layer):
        self.node_id = node_id
        self.port = port
        self.neighbors = neighbors
        self.layer = layer
        self.data_log = []
        self.update_count = 0

        # Start listening for connections
        threading.Thread(target=self.start_server, daemon=True).start()

        # Start background timer for Layer 2 (B2 -> C1, C2)
        if self.node_id == "B2":
            threading.Thread(target=self.layer2_timer, daemon=True).start()

    def start_server(self):
        # Socket setup code here...
        pass
```

Listing 1: Node Skeleton

3.2 Step 2: Core Layer Logic (Eager)

The Core layer must handle "Active" and "Eager" replication. This implies blocking the response until peers acknowledge the update.

```python
def handle_client_update(self, data):
    if self.layer == 'CORE':
        # 1. Active: Broadcast to peers (A1 -> A2, A3)
        acks = 0
        for peer in self.core_peers:
            if self.send_request(peer, "REPLICATE", data) == "ACK":
                acks += 1

        # 2. Eager: Wait for ACKs
        if acks == len(self.core_peers):
            self.commit_update(data)
            return "SUCCESS"
    return "FAIL"
```

Listing 2: Core Request Handling

3.3 Step 3: Propagation Logic (Lazy)

Nodes A2, A3, and B2 have specific triggers for propagating data downwards.

```python
def commit_update(self, data):
    self.data_log.append(data)
    self.write_log_to_file() # Requirement [cite: 36]

    # Layer 1 Trigger (A2/A3): Every 10 updates [cite: 34]
```

```
    if self.node_id in ['A2', 'A3']:
        self.update_count += 1
        if self.update_count % 10 == 0:
            target = 'B1' if self.node_id == 'A2' else 'B2'
            self.propagate_state(target)

def layer2_timer(self):
    # Layer 2 Trigger (B2): Every 10 seconds [cite: 35]
    while True:
        time.sleep(10)
        self.propagate_state('C1')
        self.propagate_state('C2')
```

Listing 3: Propagation Triggers

4 Transaction Processing

The client reads a transaction file.

- **Read-Only:** Format `30 49 69` (ends with layer ID). Can be executed on any layer[cite: 38].
- **Update:** Format `12 49.53 69`. Must always be executed on a Core node[cite: 41, 43].

5 Monitoring

A web application is required to monitor replicas in real-time via WebSockets[cite: 44].

- **Backend:** Use a lightweight server (e.g., `FastAPI` or `Flask-SocketIO`) to receive status updates from nodes.
- **Frontend:** HTML/JS Dashboard to visualize the length and content of version logs across all 7 nodes.