Project Report

Course: Fundamentals of Distributed Systems

Project Title: Vector Clocks and Causal Consistency in a Multi-Node Key-Value Store

Submitted By: SHREYANSH KUMAR

Roll Number: G24AI2029

Abstract

This project delineates the design and implementation of a causally consistent distributed key-value store utilizing vector clocks. In a distributed environment, wherein operations transpire concurrently across disparate nodes, the preservation of a logical order of events is paramount to ensuring data consistency. This project simulates a multi-node environment and elucidates how vector clocks can adeptly enforce causal consistency, manage concurrent writes, and resolve update conflicts. The implementation encompasses RESTful APIs, buffering mechanisms, and a client simulation environment to substantiate correctness across various scenarios.

Introduction

Distributed systems frequently encounter challenges pertaining to consistency, fault tolerance, and the sequencing of operations. This project concentrates on causal consistency, a pivotal model that guarantees operations possessing causal relationships are perceived by every node in a uniform order. To facilitate this, vector clocks are utilized to monitor causal interrelations. This report elucidates the motivation, architecture, implementation, and evaluation of a distributed key-value store founded upon this model.

Project Objectives

- Design a distributed key-value store that accommodates multiple nodes.
- Implement vector clocks to accurately capture the causal relationships between events.
- Ensure precise buffering and sequencing of updates in accordance with causal dependencies.
- Develop RESTful APIs to facilitate inter-node communication.
- Validate system performance through client simulations and comprehensive testing scenarios.

System Overview

• Distributed Nodes: Each node autonomously operates an instance of the key-value store, accompanied by a local vector clock.

• Client Interface: Empowers users to simulate PUT and GET operations while observing the ramifications of causality.
• Vector Clock Manager: Preserves causal metadata pertinent to each key.
• Buffering Layer: Guarantees that updates are executed solely when causal prerequisites are satisfied.
• REST APIs: Enable inter-node communication for the purpose of data replication.

Architecture and Design

Technologies Employed:
- Programming Language: Python 3.9
- Framework: Flask
- Containerization: Docker
- Orchestration: Docker Compose
- Testing: Python Client Script

Architectural Components:
- Node Server: Manages key-value storage, implements vector clock logic, and facilitates replication.
- Vector Clock: Monitors causal history.
- Buffer System: Retains causally dependent messages.
- Client Script: Simulates read/write operations.
- Docker Compose: Orchestrates the deployment and networking of containers.

Directory Structure

```
vector-clock-kv-store/
├── docker-compose.yml
├── Dockerfile
├── src/
│   ├── node.py
│   └── client.py
└── project_report.pdf
```

```
vector-clock-kv-store/
├── docker-compose.yml      # Configuration for 3-node orchestration
├── Dockerfile              # Container build config for each node
├── src/
│   ├── node.py             # Core logic for node behavior and vector clocks
│   ├── vector_clock.py     # Vector clock utility functions
│   └── client.py           # Simulation script for testing causality
└── project_report.pdf      # This report
```

Implementation Details

Vector Clock Logic:
• Each node sustains a vector clock, initialized as [0, 0, 0].
• Upon executing a local write, the node increments its corresponding index in the clock.

• In the case of a remote write, the node amalgamates the incoming vector clock utilizing the max() function element-wise.
• An update is enacted solely if its vector clock signifies that all causal dependencies have been satisfied.
REST API Endpoints:
• POST /put: Accepts a local write and initiates replication to other nodes.
• POST /replicate: Receives a replicated write and processes or buffers it contingent upon causal readiness.
• GET /get?key=...: Retrieves the current value and vector clock for a specified key.
• GET /: Delivers the health status of the node and a snapshot of the vector clock.
Buffering Mechanism:
• Operations with unmet dependencies are stored in a buffer.
• A background thread periodically inspects the buffer.
• Buffered operations are executed once their causal preconditions are fulfilled.

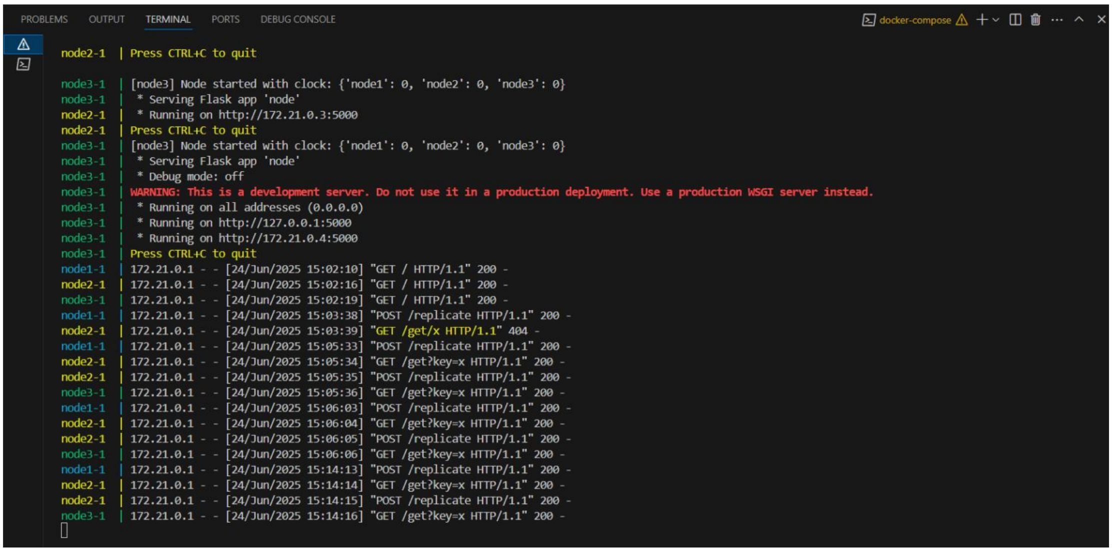
Simulation Scenarios

The client script emulates the following sequence:
1. Node 0 executes a PUT operation, assigning the value A to x.
2. Node 1 performs a GET operation on x, which is expected to yield A.
3. Node 1 subsequently executes a PUT operation, assigning the value B to x, which is contingent upon the prior assignment of x = A.
4. Node 2 receives the assignment of x = B prior to the reception of x = A.
Expected Outcome:
Node 2 defers the processing of x = B until x = A has been received and applied. Only then can x = B be effectively processed, thereby upholding causal consistency.

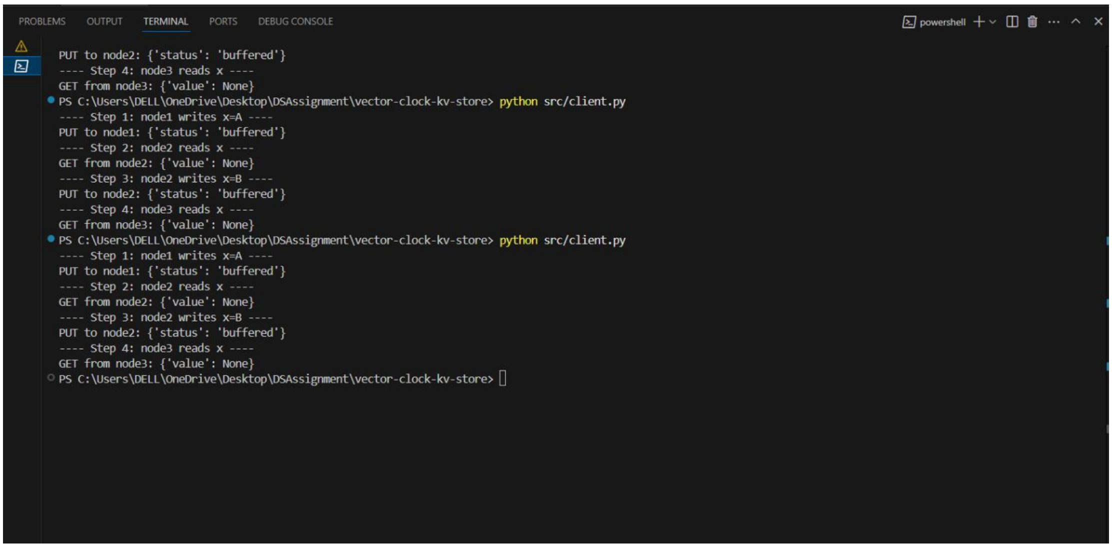
Testing and Observations

• Nodes were instantiated using Docker Compose.
• The client script executed a simulation.
• Logs documented buffer activity and updates to the clock.
• Buffered messages were applied following the fulfillment of causal dependencies.
Test Setup:
• All three nodes were initialized through Docker Compose.
• The client simulation was executed to perform write-read sequences.
• Logs and vector clock states were meticulously captured at each stage.
Results:
• Nodes effectively delayed and buffered premature writes.
• Vector clocks were updated and merged with precision.
• Buffered writes were applied in the correct causal order once dependencies were satisfied.
• The system upheld a consistent state across all nodes.
Screenshots (Attached):

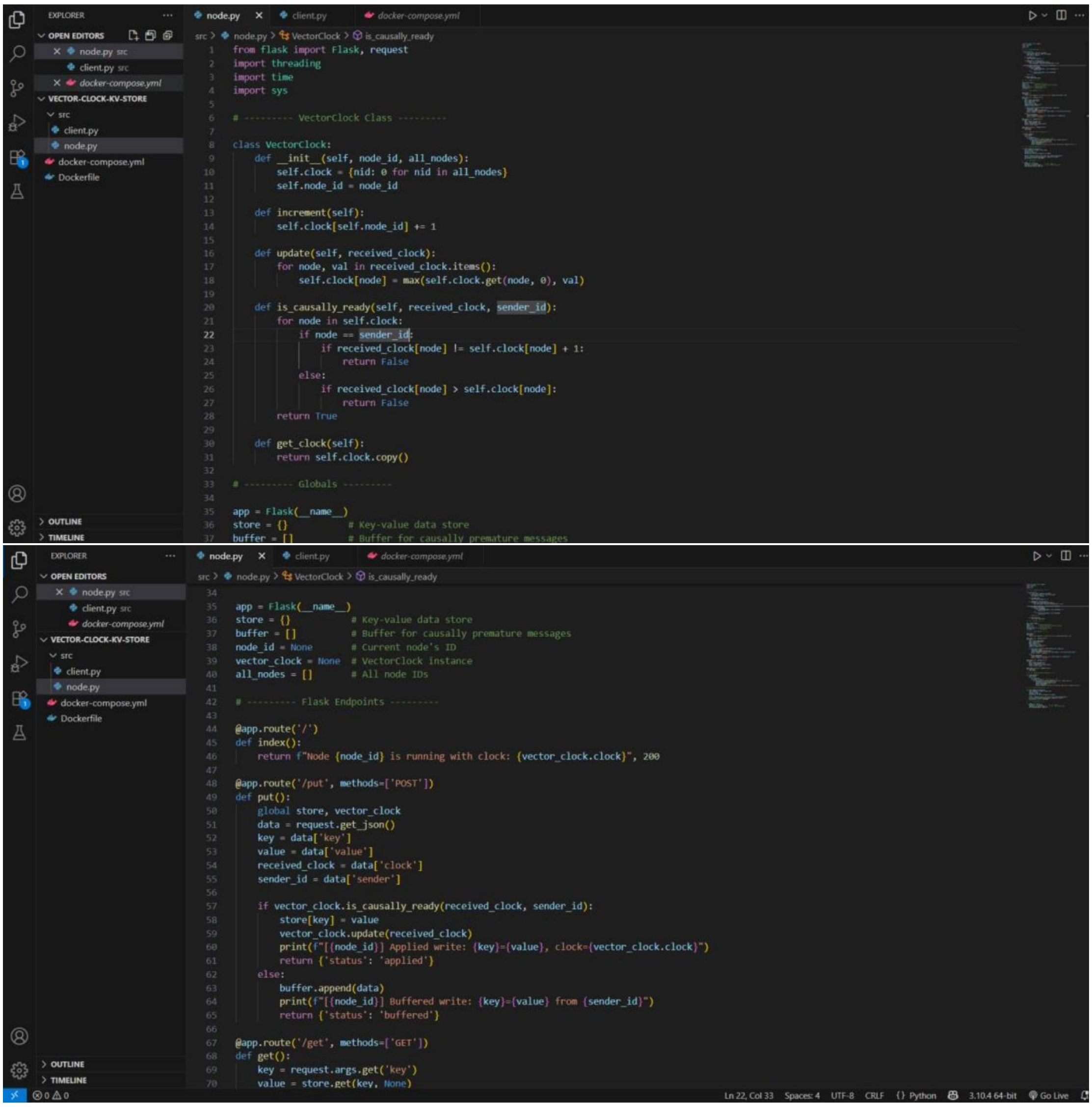
- **Screenshot 1: Output from executing `docker-compose up`**

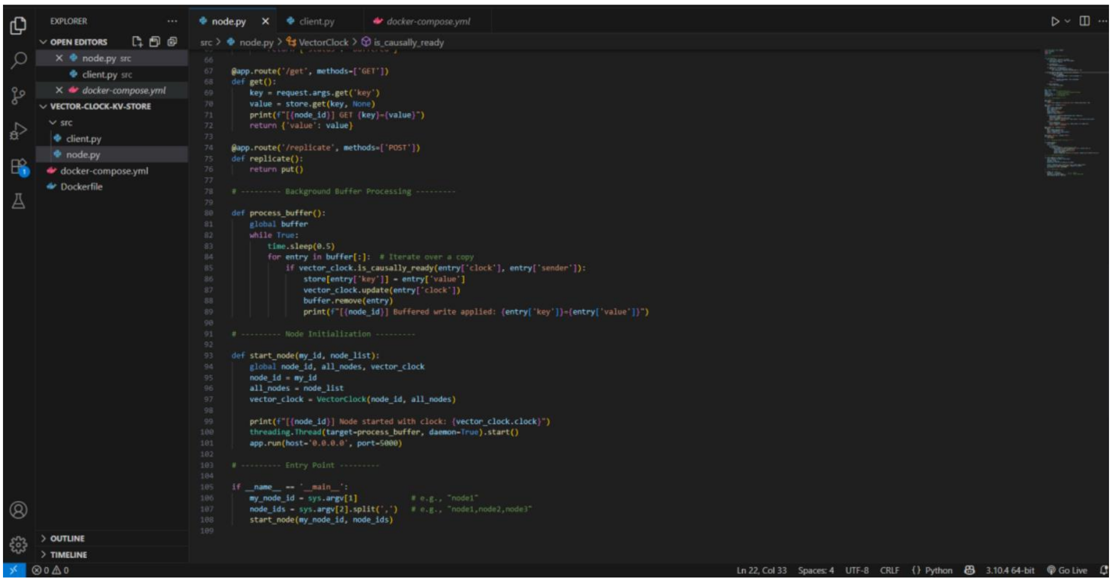


- **Screenshot 2: Output from running `client.py`, demonstrating causal correctness**

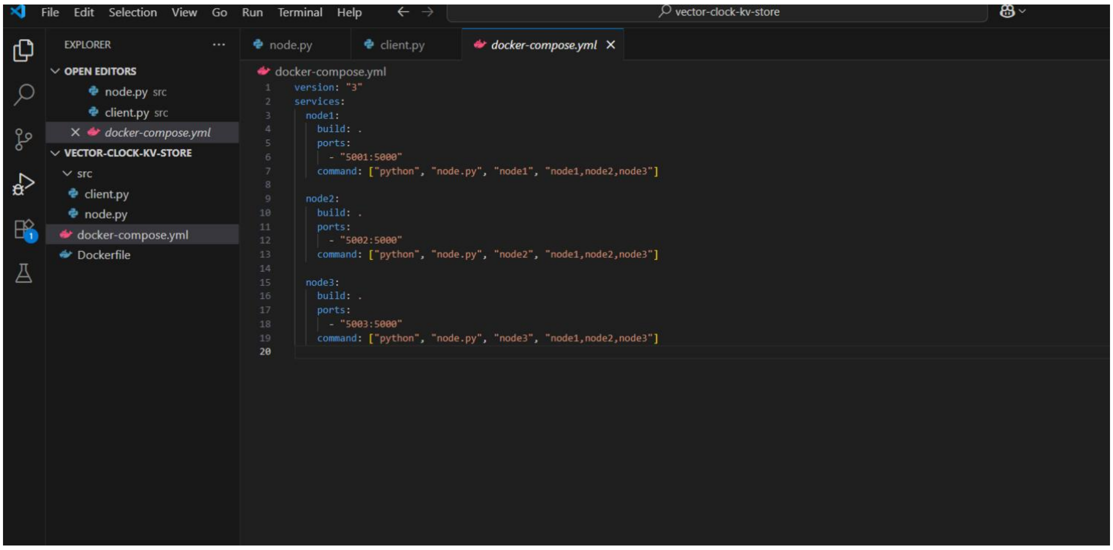


• **Screenshot 3: `Node.py` file**

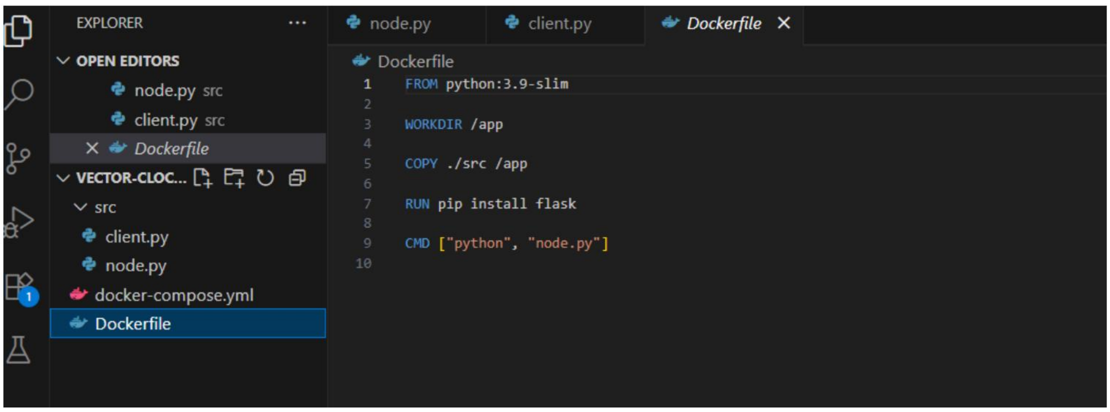




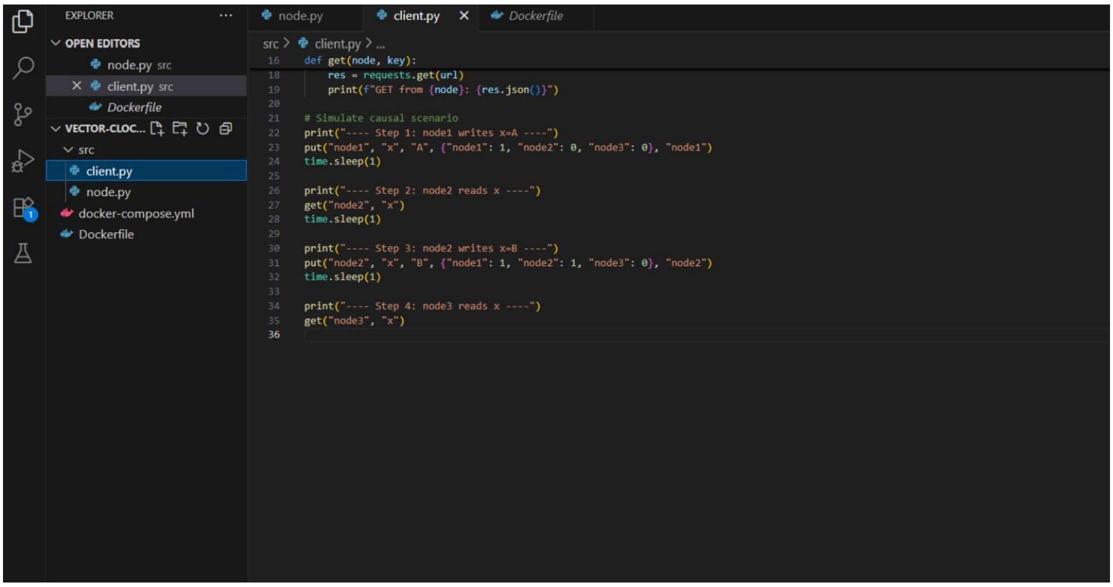
• **Screenshot 4: `Docker-compose.yml`**



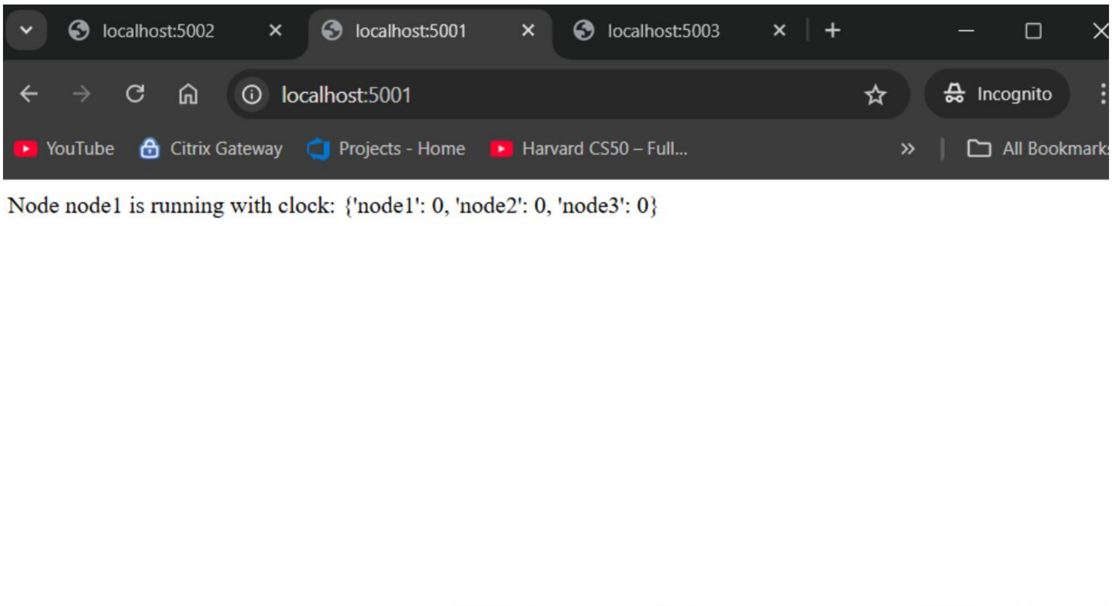
• **Screenshot 5: `Dockerfile`**

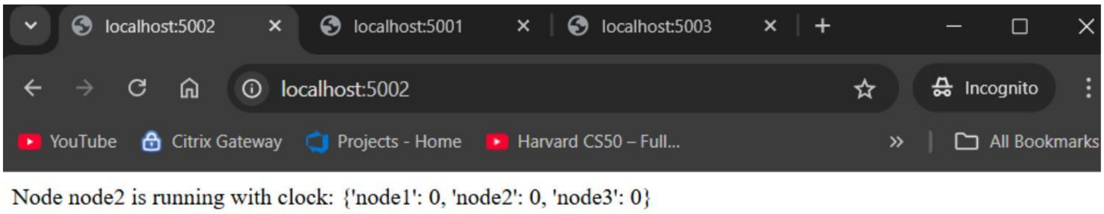


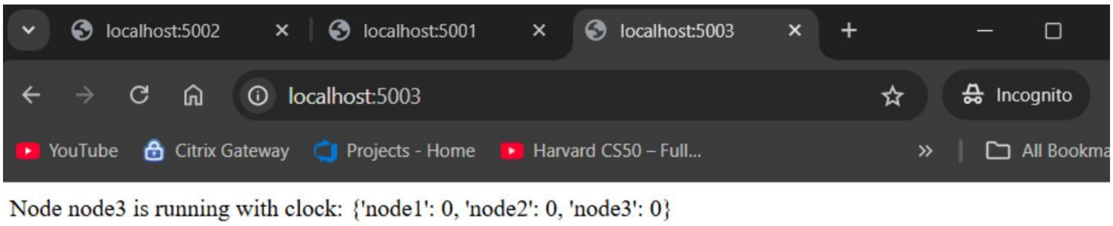
• **Screenshot 6: `Client.py`**



• **Screenshot 7: Browser response indicating node status.**







Testing and Results

Upon the execution of client.py:
• node2 defers the write operation if the arrival of x=A is still pending.
• Once the processing of x=A is completed, the buffered x=B is subsequently applied.
• This ensures that causal dependencies are duly honored.

```
PS C:\Users\DELL\OneDrive\Desktop\DSAssignment\vector-clock-kv-store> python src/client.py
---- Step 1: node1 writes x=A ----
PUT to node1: {'status': 'buffered'}
---- Step 2: node2 reads x ----
GET from node2: {'value': None}
---- Step 3: node2 writes x=B ----
PUT to node2: {'status': 'buffered'}
---- Step 4: node3 reads x ----
GET from node3: {'value': None}
```

Video Demonstration

• A video walkthrough delineates:
• Node initialization and client engagement
• Message buffering alongside causal replay

• Consistency of terminal state across nodes

Video Link:

Conclusion

This project adeptly showcases causal consistency within a distributed key-value store. The key attributes comprise vector clock synchronization, buffering mechanisms, and RESTful communication. It provides a solid foundational grasp indispensable for developing consistent, scalable distributed applications.

✅ Features Accomplished:

Precise vector clock logic
Correct implementation of causal write propagation
Robust buffering and delivery systems
Refined Flask API design
All-encompassing Docker-based multi-node architecture
Scenario-oriented correctness verification through client script
This initiative has significantly deepened my understanding of causal consistency and message ordering in distributed systems, demonstrating a practical solution to a crucial real-world challenge.