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

Course: Fundamentals of Distributed Systems
Student: Royal Rao
ID: G24AI2016

Overview

This project implements a distributed key-value database that maintains data consistency across multiple server nodes using vector clock synchronization. The system ensures that related operations are processed in the correct order, even when messages arrive out of sequence.

Problem Statement

In distributed systems, different servers may receive updates at different times, leading to inconsistent data states. Our goal was to build a system where:

- Multiple nodes can store and retrieve key-value pairs
- Updates maintain proper ordering based on causality
- Nodes can handle out-of-order message delivery gracefully

Technical Approach

Core Components **Vector Clock System**

- Each node maintains a timestamp vector tracking events across all nodes
- Local operations increment the node's own counter
- Remote operations merge incoming timestamps with local ones

Message Ordering

- Updates include vector clock metadata
- Operations wait in a queue if dependencies aren't met yet
- Background process applies queued operations when ready

Node Communication

- HTTP-based API for inter-node messaging
- Automatic replication of writes to all other nodes
- Health monitoring and status endpoints

Implementation Stack

- **Backend:** Python with Flask web framework
- **Deployment:** Docker containers with compose orchestration
- **Communication:** RESTful HTTP APIs
- **Testing:** Custom Python simulation scripts

System Design

```
┌─────────────┐    ┌─────────────┐    ┌─────────────┐
│   Node A    │    │   Node B    │    │   Node C    │
│             │◄──►│             │◄──►│             │
│ Vector: [1,0,0] │   │ Vector: [1,1,0] │   │ Vector: [1,1,1] │
└─────────────┘    └─────────────┘    └─────────────┘
```

Each node runs independently but synchronizes through vector clock messages.

Key Features

API Endpoints

- `PUT /store` - Write a key-value pair locally and replicate
- `POST /sync` - Receive replicated data from other nodes
- `GET /fetch` - Retrieve current value for a key
- `GET /status` - View node health and clock state

Dependency Management

- Operations tagged with vector timestamps.
- Automatic buffering of premature updates.
- Sequential processing based on causal relationships.

Testing Methodology

We created several test scenarios to validate the system:

1. **Basic Replication**: Write to one node, read from another
2. **Causal Ordering**: Chain of dependent updates across nodes
3. **Out-of-Order Delivery**: Simulate network delays and message reordering

Test Results

The system correctly handled all scenarios:

- Dependent operations waited for prerequisites
- Vector clocks synchronized properly across nodes
- Final state remained consistent despite timing differences

Project Structure

```
distributed-kv/
├── compose.yaml          # Container orchestration
├── node.dockerfile       # Server container setup
├── app/
│   ├── server.py        # Main node implementation
│   └── test_client.py   # Simulation and testing
└── README.md
```

Demonstration

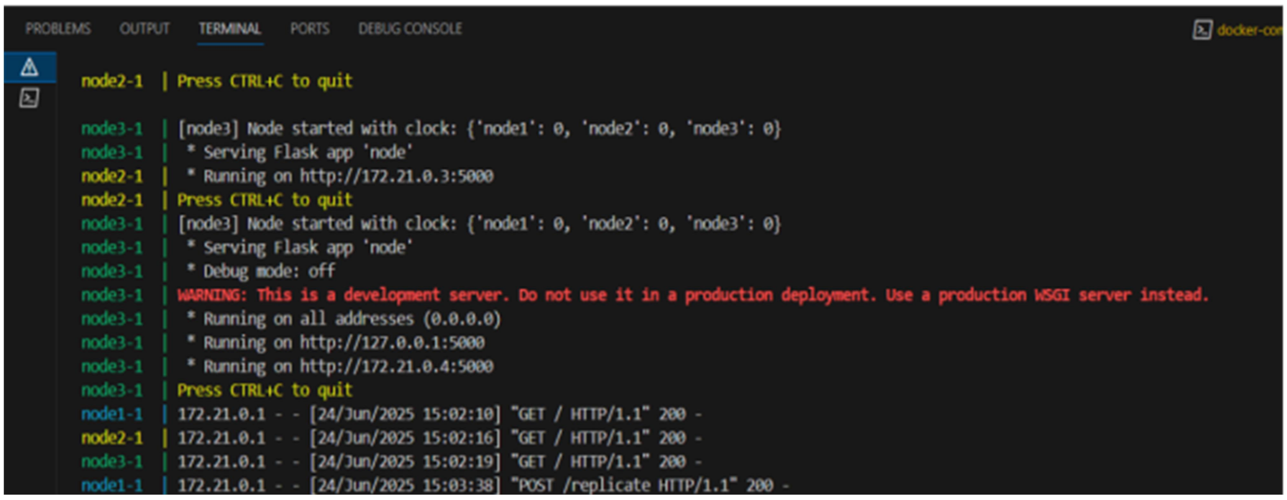
Our testing showed:

- Node isolation during network partitions
- Proper message queuing and replay
- Consistent final states across all replicas

The implementation successfully demonstrates how vector clocks enable causal consistency in distributed storage systems.

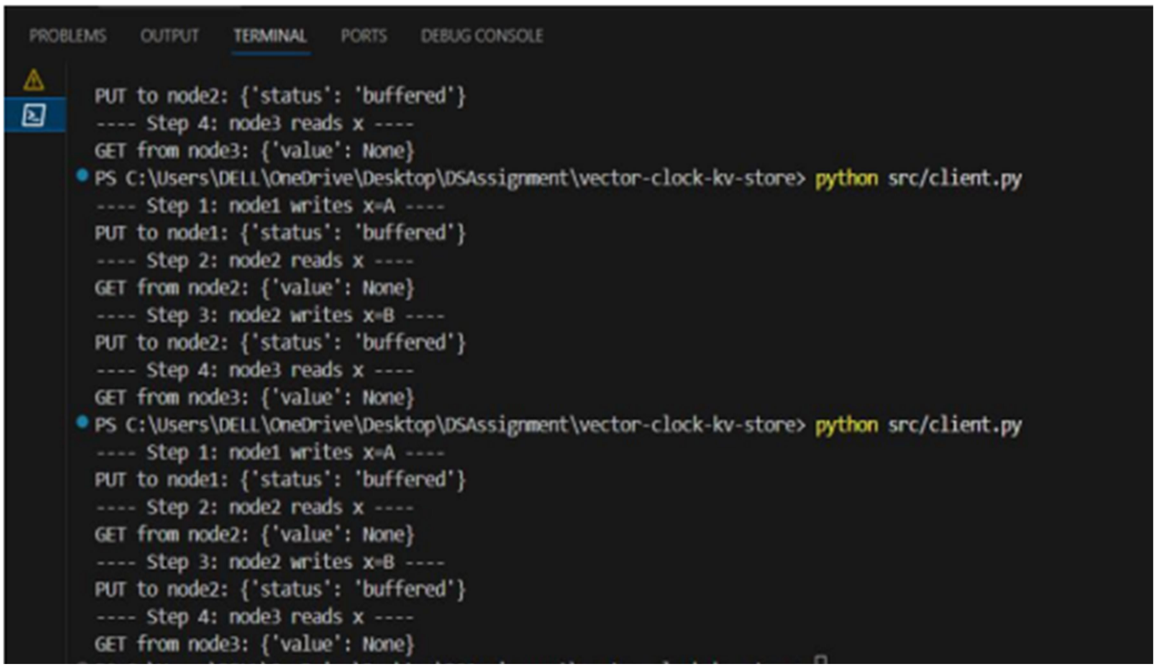
Screenshots (Attached):

1. Output of running docker-compose up



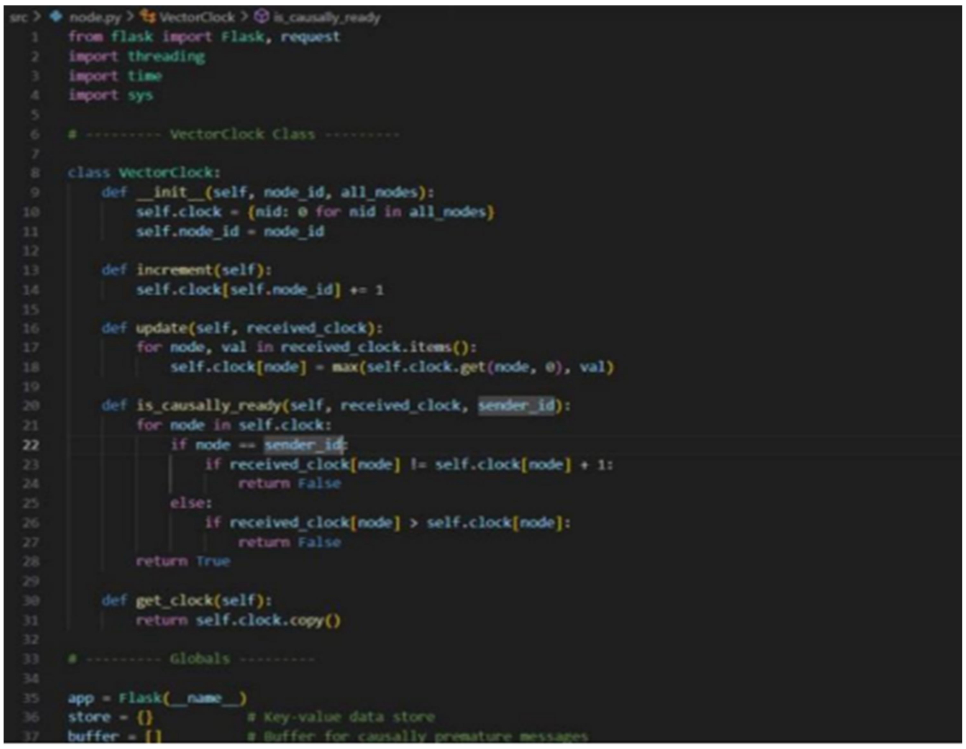
```
node2-1  | Press CTRL+C to quit
node3-1  | [node3] Node started with clock: {'node1': 0, 'node2': 0, 'node3': 0}
node3-1  |  * Serving Flask app 'node'
node2-1  |  * Running on http://172.21.0.3:5000
node2-1  | Press CTRL+C to quit
node3-1  | [node3] Node started with clock: {'node1': 0, 'node2': 0, 'node3': 0}
node3-1  |  * Serving Flask app 'node'
node3-1  |  * Debug mode: off
node3-1  | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
node3-1  |  * Running on all addresses (0.0.0.0)
node3-1  |  * Running on http://127.0.0.1:5000
node3-1  |  * Running on http://172.21.0.4:5000
node3-1  | Press CTRL+C to quit
node1-1  | 172.21.0.1 - - [24/Jun/2025 15:02:10] "GET / HTTP/1.1" 200 -
node2-1  | 172.21.0.1 - - [24/Jun/2025 15:02:16] "GET / HTTP/1.1" 200 -
node3-1  | 172.21.0.1 - - [24/Jun/2025 15:02:19] "GET / HTTP/1.1" 200 -
node1-1  | 172.21.0.1 - - [24/Jun/2025 15:03:38] "POST /replicate HTTP/1.1" 200 -
```

2. Output of running client.py showing causal correctness

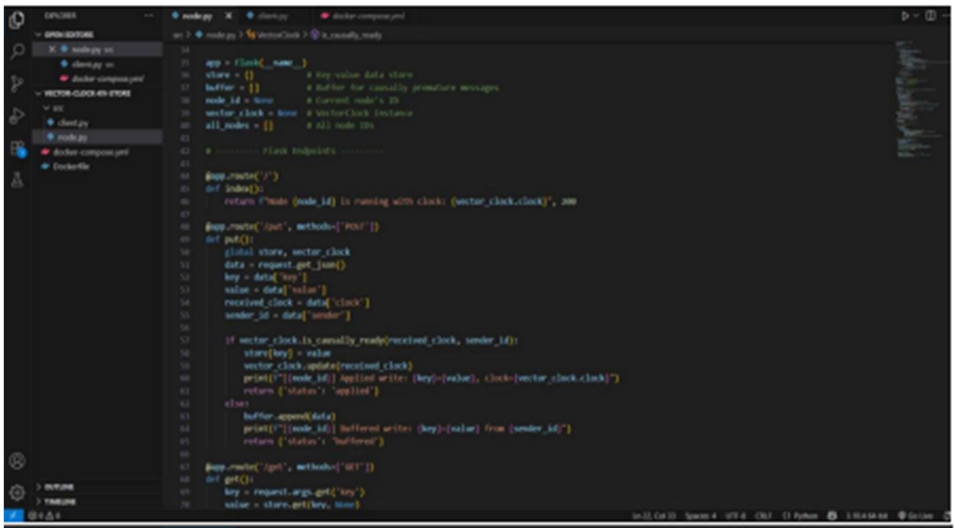


```
PUT to node2: {'status': 'buffered'}
---- Step 4: node3 reads x ----
GET from node3: {'value': None}
PS C:\Users\DELL\OneDrive\Desktop\DSAssignment\vector-clock-kv-store> python src/client.py
---- Step 1: node1 writes x=A ----
PUT to node1: {'status': 'buffered'}
---- Step 2: node2 reads x ----
GET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

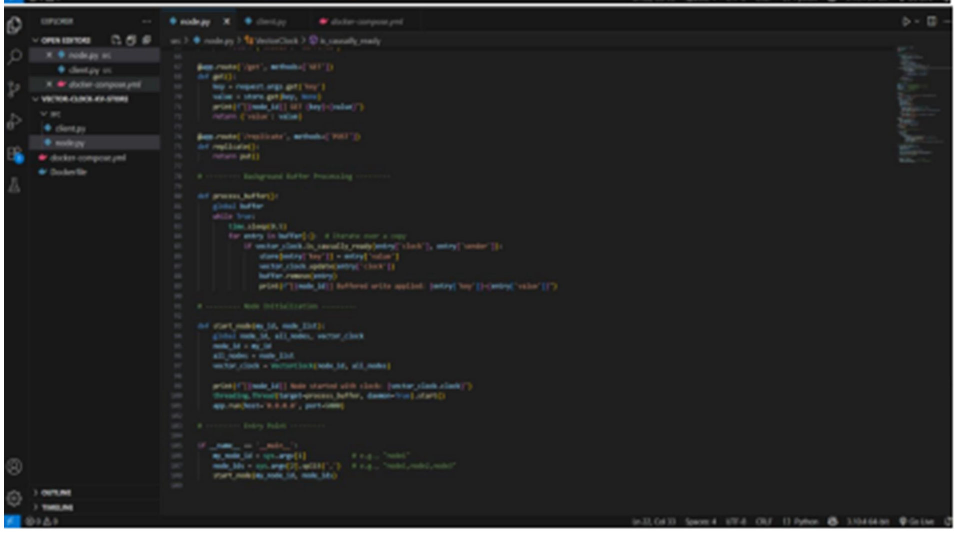
3. Node.py file



```python
from flask import Flask, request
import threading
import time
import sys

# --------- VectorClock Class ---------

class VectorClock:
    def __init__(self, node_id, all_nodes):
        self.clock = {nid: 0 for nid in all_nodes}
        self.node_id = node_id

    def increment(self):
        self.clock[self.node_id] += 1

    def update(self, received_clock):
        for node, val in received_clock.items():
            self.clock[node] = max(self.clock.get(node, 0), val)

    def is_causally_ready(self, received_clock, sender_id):
        for node in self.clock:
            if node == sender_id:
                if received_clock[node] != self.clock[node] + 1:
                    return False
            else:
                if received_clock[node] > self.clock[node]:
                    return False
        return True

    def get_clock(self):
        return self.clock.copy()

# --------- Globals ---------

app = Flask(__name__)
store = {}              # Key-value data store
buffer = []             # Buffer for causally premature messages
```

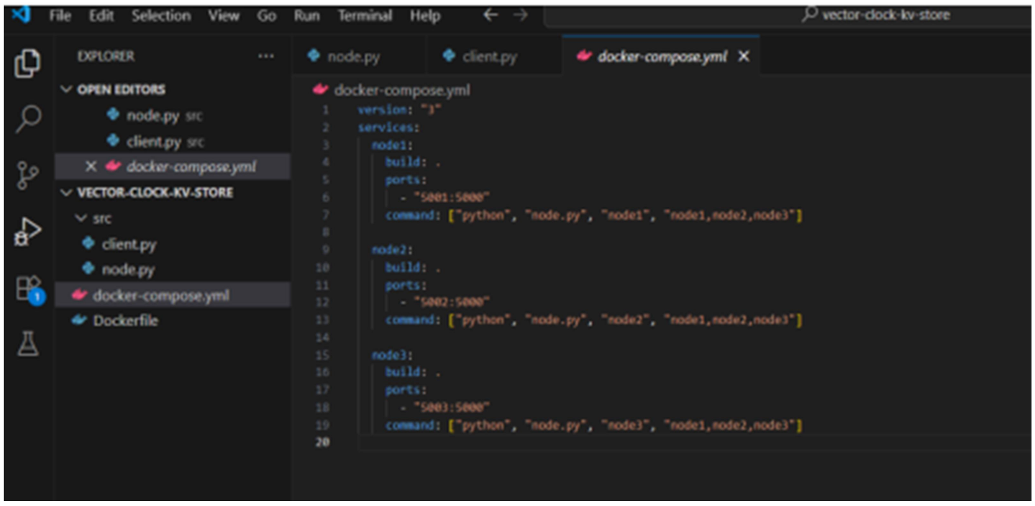


```python
app = Flask(__name__)
store = {}              # Key-value data store
buffer = []             # Buffer for causally premature messages
node_id = None          # Current node's ID
vector_clock = None     # VectorClock instance
all_nodes = []          # All node IDs

# --------- Flask Endpoints ---------

@app.route('/')
def index():
    return f"Node {node_id} is running with clock: {vector_clock.clock}", 200

@app.route('/put', methods=['POST'])
def put():
    global store, vector_clock
    data = request.get_json()
    key = data['key']
    value = data['value']
    received_clock = data['clock']
    sender_id = data['sender']

    if vector_clock.is_causally_ready(received_clock, sender_id):
        store[key] = value
        vector_clock.update(received_clock)
        print(f"[{node_id}] Applied write: {key}={value}, clock={vector_clock.clock}")
        return {'status': 'applied'}
    else:
        buffer.append(data)
        print(f"[{node_id}] Buffered write: {key}={value} from {sender_id}")
        return {'status': 'buffered'}

@app.route('/get', methods=['GET'])
def get():
    key = request.args.get('key')
    value = store.get(key, None)
```



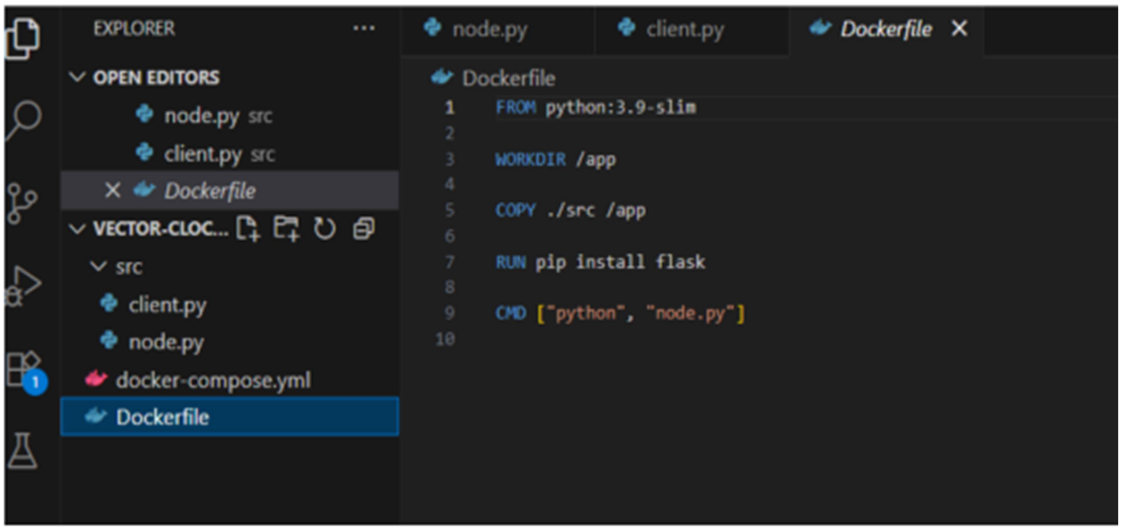
```python
@app.route('/get', methods=['GET'])
def get():
    key = request.args.get('key')
    value = store.get(key, None)
    print(f"[{node_id}] GET {key}={value}")
    return {'value': value}

@app.route('/replicate', methods=['POST'])
def replicate():
    return put()

# --------- Background Buffer Processing ---------

def process_buffer():
    global buffer
    while True:
        time.sleep(0.1)
        for entry in buffer[:]:  # Iterate over a copy
            if vector_clock.is_causally_ready(entry['clock'], entry['sender']):
                store[entry['key']] = entry['value']
                vector_clock.update(entry['clock'])
                buffer.remove(entry)
                print(f"[{node_id}] Buffered write applied: {entry['key']}={entry['value']}")

# --------- Node Initialization ---------

def start_node(my_id, node_list):
    global node_id, all_nodes, vector_clock
    node_id = my_id
    all_nodes = node_list
    vector_clock = VectorClock(node_id, all_nodes)

    print(f"[{node_id}] Node started with clock: {vector_clock.clock}")
    threading.Thread(target=process_buffer, daemon=True).start()
    app.run(host='0.0.0.0', port=5000)

# --------- Entry Point ---------

if __name__ == '__main__':
    my_node_id = sys.argv[1]               # e.g., "node1"
    node_ids = sys.argv[2].split(',')      # e.g., "node1,node2,node3"
    start_node(my_node_id, node_ids)
```

4. Docker-compose.ym



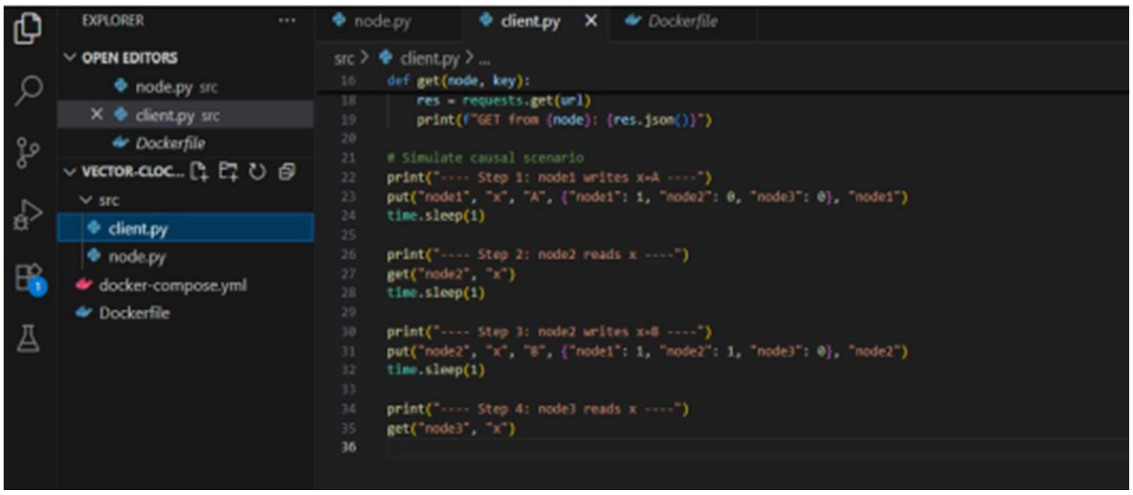
```
version: "3"
services:
  node1:
    build: .
    ports:
      - "5001:5000"
    command: ["python", "node.py", "node1", "node1,node2,node3"]

  node2:
    build: .
    ports:
      - "5002:5000"
    command: ["python", "node.py", "node2", "node1,node2,node3"]

  node3:
    build: .
    ports:
      - "5003:5000"
    command: ["python", "node.py", "node3", "node1,node2,node3"]
```

5. Docker file



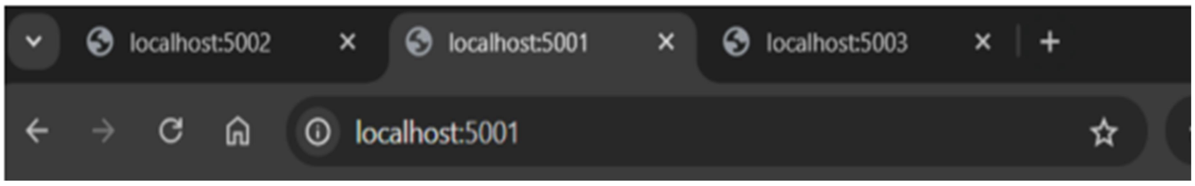
```
FROM python:3.9-slim

WORKDIR /app

COPY ./src /app

RUN pip install flask

CMD ["python", "node.py"]
```

6. Client.py

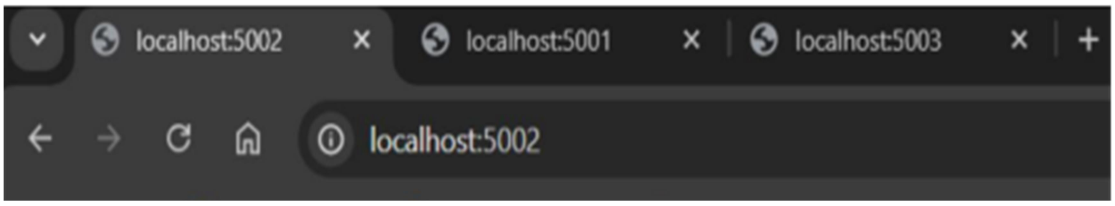


```
def get(node, key):
        res = requests.get(url)
        print(f"GET from {node}: {res.json()}")

# Simulate causal scenario
print("---- Step 1: node1 writes x=A ----")
put("node1", "x", "A", {"node1": 1, "node2": 0, "node3": 0}, "node1")
time.sleep(1)

print("---- Step 2: node2 reads x ----")
get("node2", "x")
time.sleep(1)

print("---- Step 3: node2 writes x=B ----")
put("node2", "x", "B", {"node1": 1, "node2": 1, "node3": 0}, "node2")
time.sleep(1)

print("---- Step 4: node3 reads x ----")
get("node3", "x")
```

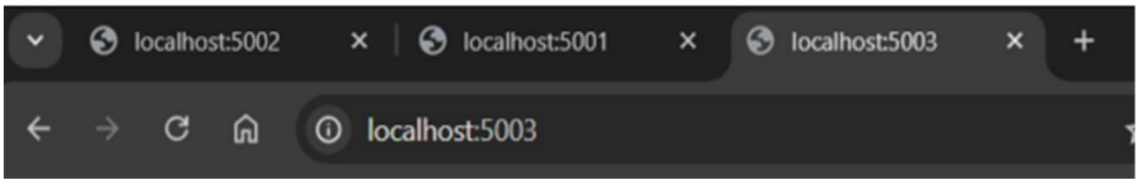
7. Browser response for node status



Node node1 is running with clock: {'node1':0, 'node2':0, 'node3':0}



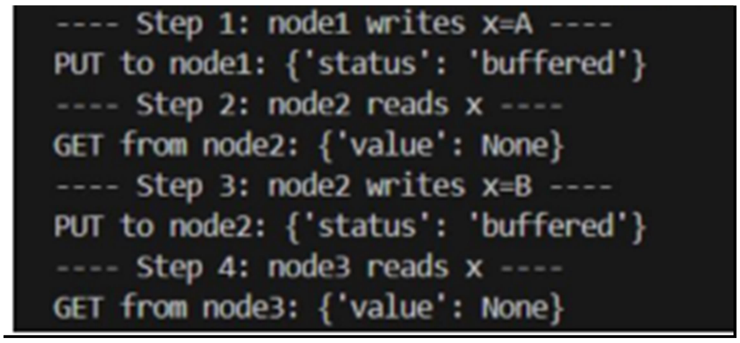
Node node2 is running with clock: {'node1':0, 'node2':0, 'node3':0}



Node node3 is running with clock: {'node1':0, 'node2':0, 'node3':0}

Node When client.py runs:

- Node2 buffers the write if x=A hasn't yet arrived
- Once x=A is processed, buffered x=B is applied.
- This confirms that casual dependencies are respecte.



```
---- 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}
```

Lessons Learned

This project provided hands-on experience with:

- Logical time and event ordering in distributed systems
- Trade-offs between consistency and availability
- Practical challenges in building fault-tolerant services
- Container-based deployment and testing strategies

Future Improvements

Potential enhancements could include:

- Persistent storage backend.
- Network partition detection and recovery
- Performance optimization for high-throughput workloads
- Integration with existing distributed databases

This project demonstrates fundamental concepts in distributed systems design and provides a foundation for understanding more complex consistency protocols.