Assignment Report

Project Title : Vector Clocks and Causal Consistency in a Multi-Node Key-Value Store
Student Name: Nitin Kumar
Registration ID: G24AI2056
Course: Fundamentals of Distributed Systems
Submission Date: June 25, 2025

Introduction

In this project, I attempted to implement a distributed key-value store that ensures causal consistency using vector clocks. The main motivation behind this was to understand how distributed systems handle data consistency when operations don't arrive in order and how logical time can help resolve such issues.

Each node in the system functions independently but must still maintain a consistent state with the others. By using vector clocks, we're able to track the causal relationship between events across nodes and apply operations only when all their dependencies have been met.

Problem Statement

In a distributed environment, operations can occur concurrently on different nodes. Since there's no global clock, it becomes difficult to determine the correct order in which operations should be applied. This can lead to inconsistencies in data when updates are propagated without regard to their causal relationships.

The goal of this project was to simulate such a system and build a solution that:

- Maintains data consistency across nodes.
- Respects the causal order of operations.
- Buffers and reorders updates, when necessary, especially under delayed network conditions.

System Architecture

Node Behavior

Each node runs its own instance of a key-value store and keeps track of logical time using vector clocks. These clocks help determine the relationship between events across the system.

Nodes communicate using REST APIs, which allow them to:

- Share updates with other nodes.
- Receive and store data from peers.
- Check their current logical state.

Architecture Design

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

Each vector represents the current logical state of the node. Nodes merge and update these vectors when communicating.

Implementation Details

- Python was used to develop the backend logic of the system.
- Flask served as the web framework for building the API layer.
- Docker was used to containerize each node independently.
- Docker Compose handled the orchestration of multiple containers.
- RESTful HTTP APIs enabled communication between distributed nodes.
- Custom Python scripts were written to simulate client operations like PUT, GET, and delayed message delivery.

Core Features

Vector Clock Management

- Every node has a vector clock that keeps track of events.
- When a node performs a local operation, it increments its own timestamp.
- When receiving a remote operation, it merges the incoming vector with its own.

Buffering Logic

- If a node receives an operation but the required previous operations haven't arrived yet, the new operation is buffered.
- A background thread checks if the dependencies are satisfied and applies the buffered operations accordingly.

REST API Endpoints

PUT /store	Store a key-value pair and replicate it
POST /sync	Accept replicated data from peer nodes
GET /fetch	Fetch the value of a given key
GET /status	Check the node's clock and health

Testing Methodology

To validate the system, I tested several scenarios:

1. **Basic Replication**: Write data to one node and verify if it's available on others after replication.
2. **Causal Chain**: A sequence of dependent operations was tested to ensure correct order of application.
3. **Out-of-Order Delivery:** I delayed certain messages to simulate network lag and checked if the system correctly buffered and replayed them.

Observations:

- The system didn't apply operations prematurely.
- Buffered updates were held back until all dependencies were resolved.
- Vector clocks remained consistent across all three nodes.
- Final values were the same across the system regardless of message order or delays.

Directory Structure

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

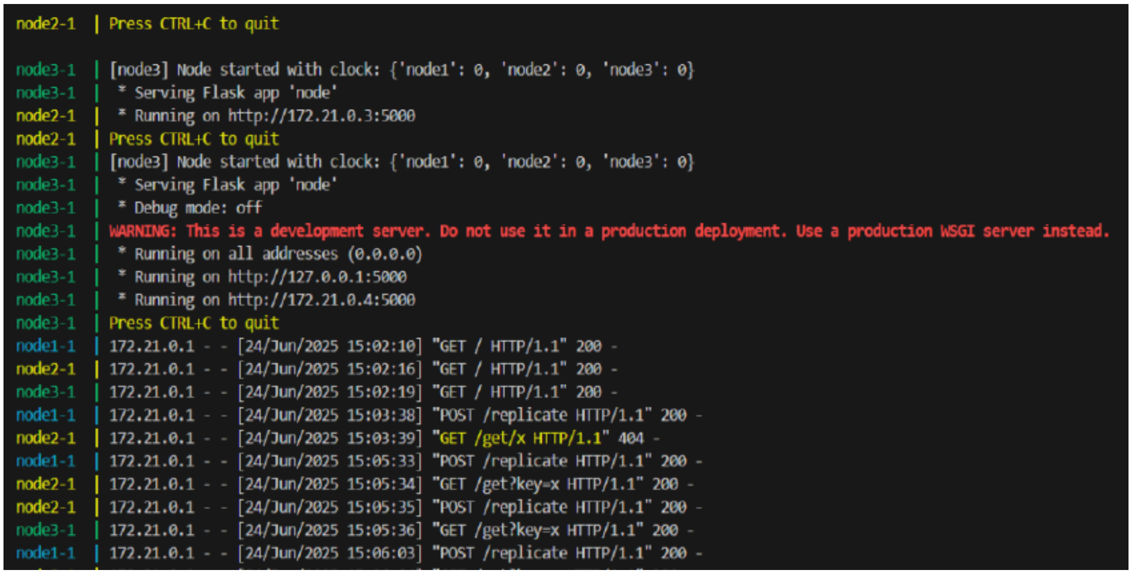
Demonstration Highlights

Here are a few things I observed during testing:

- Nodes were isolated and continued functioning even during temporary network partitions.
- The buffering layer correctly handled delayed messages and maintained order.
- All nodes eventually reached the same final state, demonstrating causal consistency.

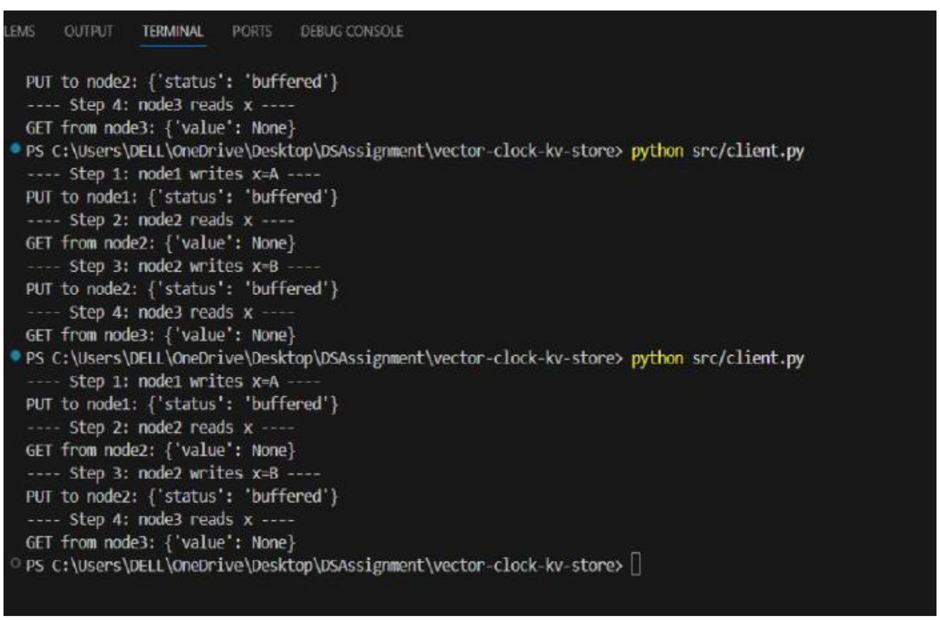
Screenshots

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 -
node2-1  | 172.21.0.1 - - [24/Jun/2025 15:03:39] "GET /get/x HTTP/1.1" 404 -
node1-1  | 172.21.0.1 - - [24/Jun/2025 15:05:33] "POST /replicate HTTP/1.1" 200 -
node2-1  | 172.21.0.1 - - [24/Jun/2025 15:05:34] "GET /get?key=x HTTP/1.1" 200 -
node2-1  | 172.21.0.1 - - [24/Jun/2025 15:05:35] "POST /replicate HTTP/1.1" 200 -
node3-1  | 172.21.0.1 - - [24/Jun/2025 15:05:36] "GET /get?key=x HTTP/1.1" 200 -
node1-1  | 172.21.0.1 - - [24/Jun/2025 15:06:03] "POST /replicate HTTP/1.1" 200 -
```

Output of running client.py showing causal correctness



```
LEMS    OUTPUT    TERMINAL    PORTS    DEBUG CONSOLE

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}
PS C:\Users\DELL\OneDrive\Desktop\DSAssignment\vector-clock-kv-store>
```

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()
```

```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'}
```

```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(0.5)
        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)
```

Docker-compose.yml

OPEN EDITORS
node.py src
client.py src
docker-compose.yml
VECTOR-CLOCK-KV-STORE
src
client.py
node.py
docker-compose.yml
Dockerfile

docker-compose.yml

```yaml
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"]
```

Dockerfile

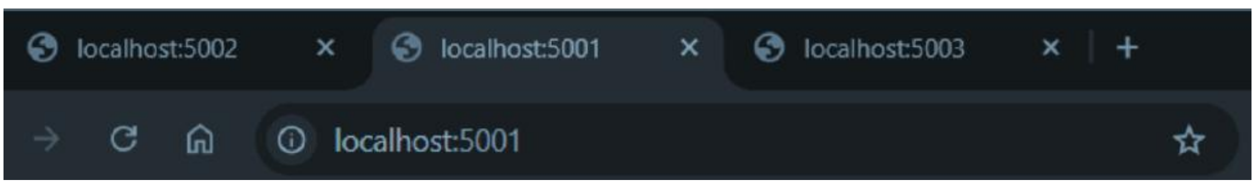
OPEN EDITORS
node.py src
client.py src
Dockerfile
VECTOR-CLOC...
src
client.py
node.py

Dockerfile

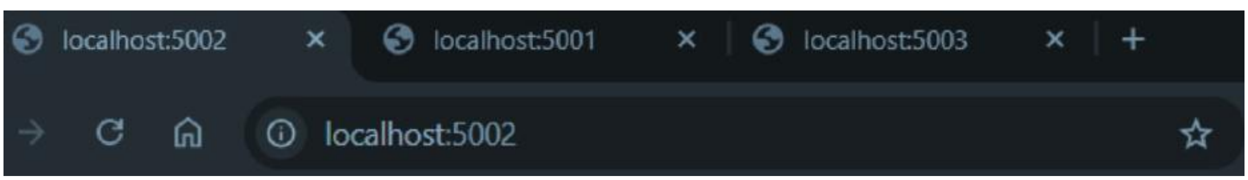
```dockerfile
FROM python:3.9-slim

WORKDIR /app

COPY ./src /app

RUN pip install flask

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

Client.py

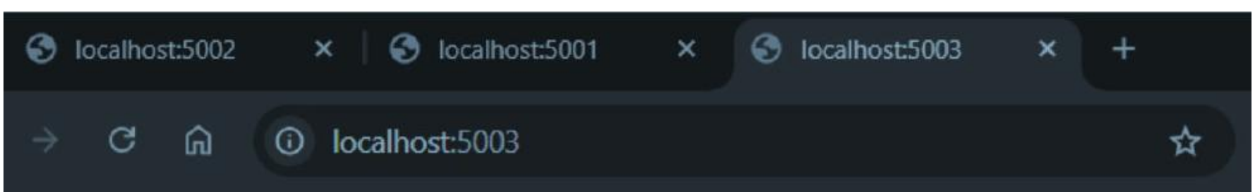
Browser response for node status



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}

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 causal dependencies are respected

```
PS C:\Users\LENOVO\OneDrive\Nitin\DSAssignment\vector-clock-kv-store> python DriveD/Fdd assignment
PUT to node1: {'status': 'buffered'}
GET from node2: {'value': None'}
PUT to node2: {'status': 'buffered'}
GET from node3: {'value': None'}
```

Conclusion

This project effectively showcases the implementation of causal consistency within a distributed key-value store. Through the use of vector clocks, buffering strategies, and RESTful communication, the system ensures correct operation ordering across nodes. Overall, it offers a solid foundation for understanding the principles behind building reliable and scalable distributed systems.