

Project Report: Vector Clocks and Causal Ordering in Distributed Key-Value Store

Course: Distributed Systems

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

Technologies Used: Python, Docker, Docker Compose

Submitted by: Harshita Gupta

Roll No: g24ai2017

Date: 25/06/25

Github Repo: <https://github.com/Harshita217/Harshita-DS-Assignment/tree/main>

1. Project Objective

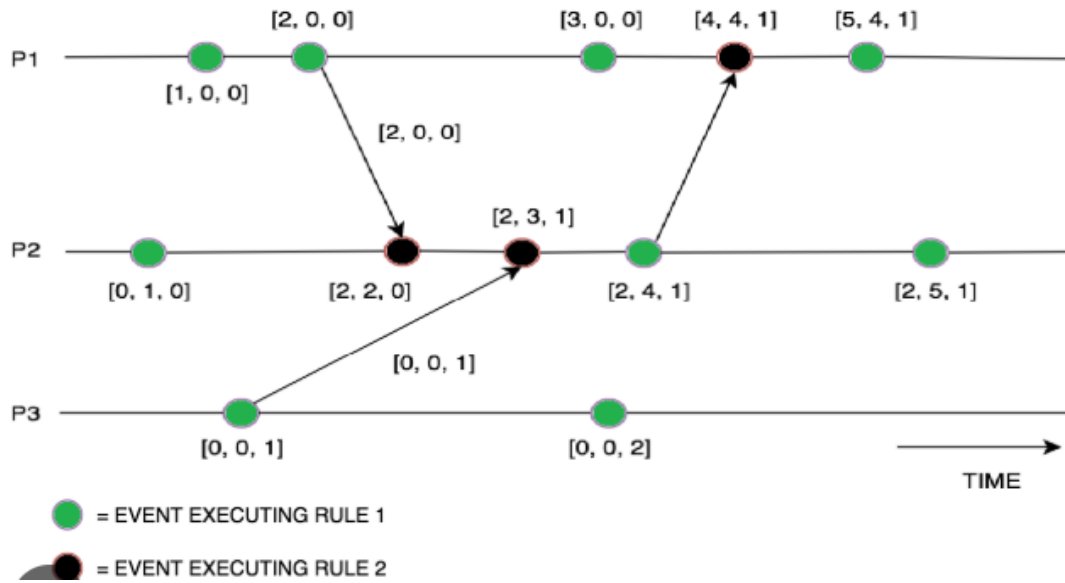
The goal of this project is to build a distributed key-value store with causal consistency using Vector Clocks. This ensures that operations across nodes respect causal relationships, preventing scenarios where dependent events are processed out of order.

2. System Architecture

The system consists of:

- Three Nodes: Each running an instance of the key-value store with its own vector clock.
- Client Application: Allows interaction with the nodes for testing.
- Vector Clocks: Track causal dependencies between events across nodes.
- Buffered Messages: Writes that arrive before their causal dependencies are buffered until they can be safely applied.

Architecture Diagram:



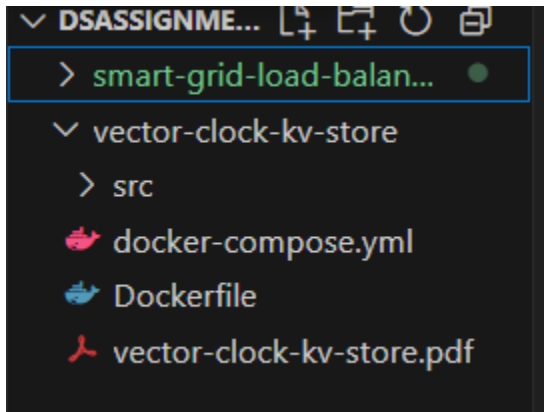
3. Technology Stack

Component	Technology
Programming	Python (3.9)
Web Framework	Flask
Containerization	Docker
Orchestration	Docker Compose
Testing	Custom client script

3. Architecture:

- Nodes: Each node runs a Flask server in its own Docker container. It maintains a local key-value store and a vector clock.
- Vector **Clock**: Used to capture and check causal dependencies between events.
- Communication: Writes are replicated to other nodes. Each write carries its vector clock.
- Buffering: If a node receives a write that is not causally ready, it buffers the write until the dependencies are met.
- Client: A Python script simulates a causal scenario and verifies correctness.

4. Directory Structure



5. Key Implementation Highlights

VectorClock Class

- Maintains a dictionary of counters for all nodes.
- `increment()`, `update()` and `is_causally_ready()` ensure correct causality logic.

Flask Endpoints

- `/put`: Accepts writes and applies or buffers them.
- `/replicate`: Alias to `/put` used for remote writes.
- `/get`: Reads values from the local store.
- `/`: Health check route to show node clock and status.

Buffering Mechanism

- Runs in a background thread.
- Periodically checks if buffered messages are ready for application.

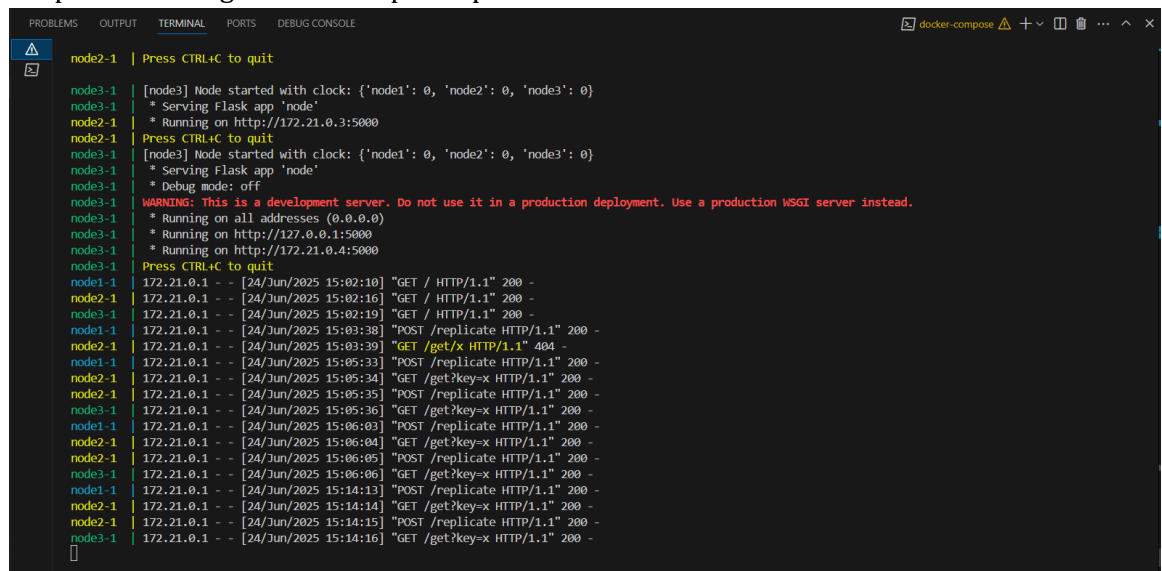
Client Script

- Simulates this scenario:
 1. Node1 writes x = A
 2. Node2 reads x
 3. Node2 writes x = B (after reading A)
 4. Node3 reads x

→ This verifies that x = B is only applied after x = A.

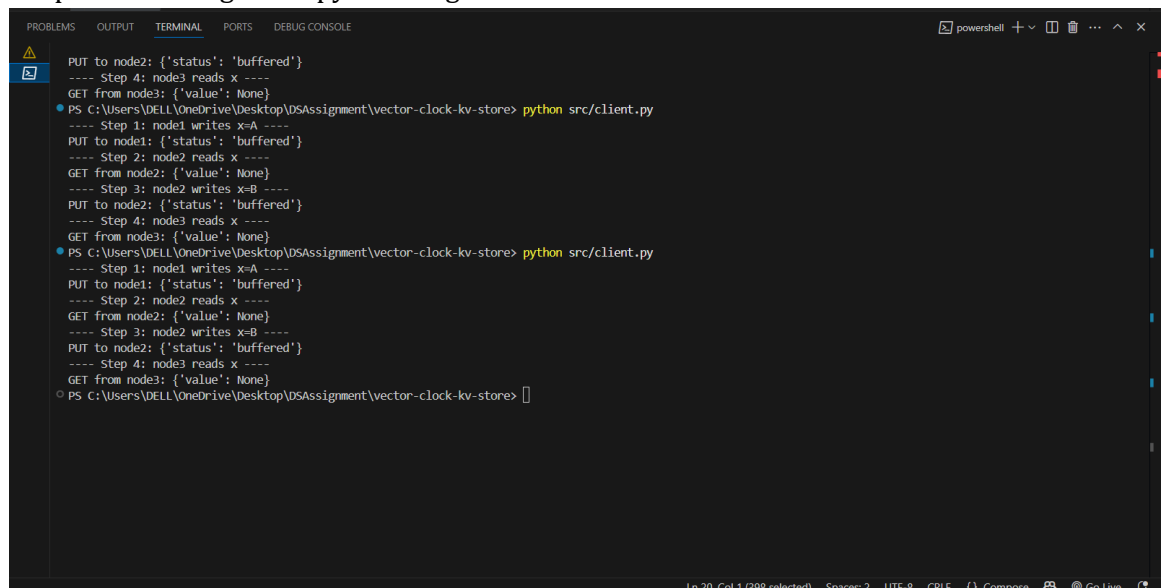
6. Screenshots

- Output of running docker-compose up



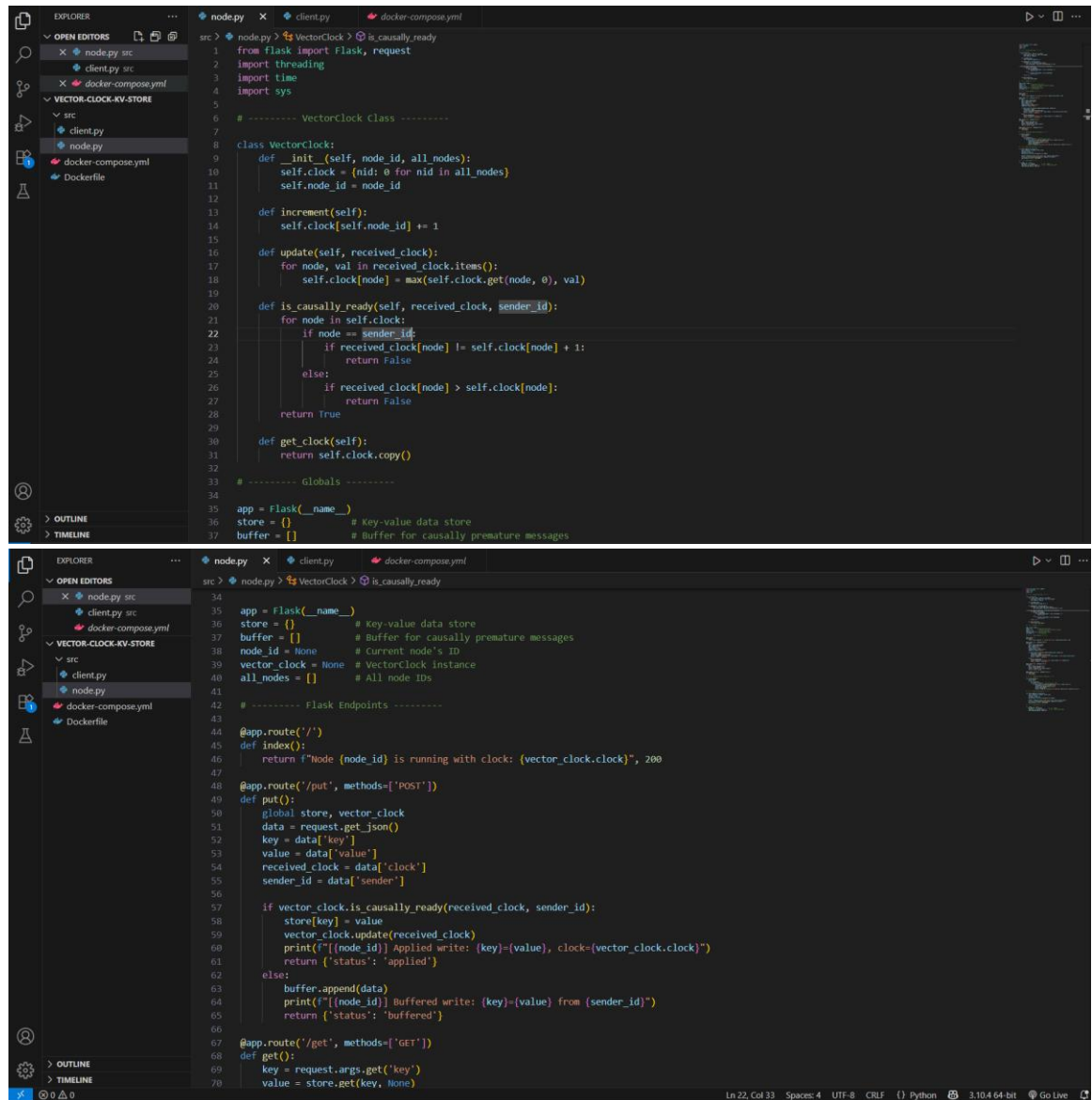
```
node2-1 | Press CTRL+C to quit
node3-1 | [node3] Node started with clock: {'node1': 0, 'node2': 0, 'node3': 0}
node3-1 | * Serving Flask app 'node'
node2-1 | * Running on http://172.21.0.3:5000
node2-1 | Press CTRL+C to quit
node3-1 | [node3] Node started with clock: {'node1': 0, 'node2': 0, 'node3': 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 -
node2-1 | 172.21.0.1 - - [24/Jun/2025 15:06:04] "GET /get?key=x HTTP/1.1" 200 -
node2-1 | 172.21.0.1 - - [24/Jun/2025 15:06:05] "POST /replicate HTTP/1.1" 200 -
node3-1 | 172.21.0.1 - - [24/Jun/2025 15:06:06] "GET /get?key=x HTTP/1.1" 200 -
node1-1 | 172.21.0.1 - - [24/Jun/2025 15:14:13] "POST /replicate HTTP/1.1" 200 -
node2-1 | 172.21.0.1 - - [24/Jun/2025 15:14:14] "GET /get?key=x HTTP/1.1" 200 -
node2-1 | 172.21.0.1 - - [24/Jun/2025 15:14:15] "POST /replicate HTTP/1.1" 200 -
node3-1 | 172.21.0.1 - - [24/Jun/2025 15:14:16] "GET /get?key=x HTTP/1.1" 200 -
```

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

- Node.py file



The image displays two screenshots of a VS Code editor window, showing the implementation of a VectorClock in Python and its integration with a Flask web application.

Top Screenshot: The editor shows the `node.py` file. The code defines a `VectorClock` class with the following methods:

- `__init__(self, node_id, all_nodes)`: Initializes the clock for a specific node.
- `increment(self)`: Increments the clock value for the current node.
- `update(self, received_clock)`: Updates the clock with the maximum value from the received clock.
- `is_causally_ready(self, received_clock, sender_id)`: Checks if a message from the sender is causally ready to be applied.
- `get_clock(self)`: Returns a copy of the current clock.

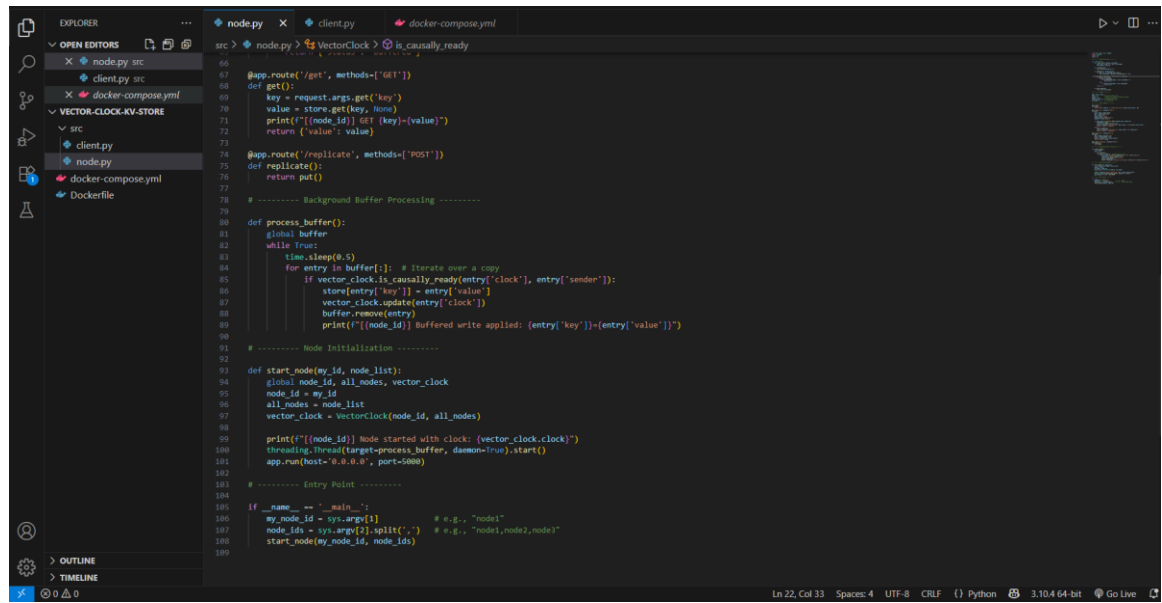
The code also includes a Flask application setup with a `store` dictionary and a `buffer` list for handling messages.

Bottom Screenshot: The editor shows the `node.py` file, focusing on the Flask endpoints and global variables.

- `app = Flask(__name__)`: Creates the Flask application.
- `store = {}`: A dictionary to store key-value pairs.
- `buffer = []`: A list to buffer messages that are not yet causally ready.
- `node_id = None`: The current node's ID.
- `vector_clock = None`: The VectorClock instance.
- `all_nodes = []`: A list of all node IDs.

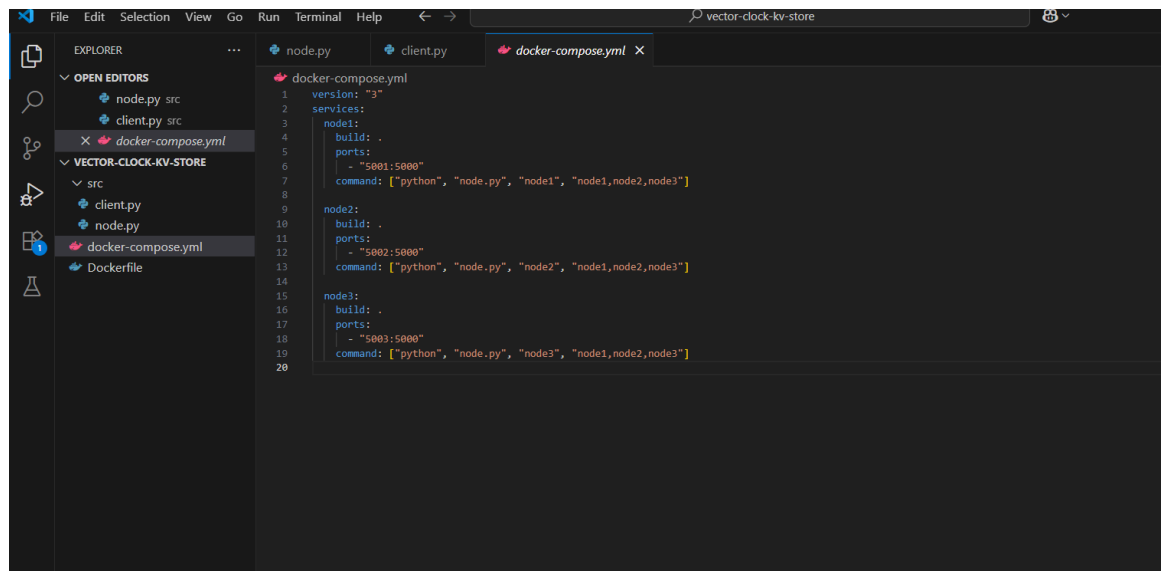
The endpoints are defined as follows:

- `index()`: Returns a message indicating the node is running with its current clock.
- `put()`: A POST endpoint that receives a key-value pair and a clock. It checks if the message is causally ready. If yes, it updates the store and the clock. If no, it buffers the message.
- `get()`: A GET endpoint that returns the value for a given key from the store.



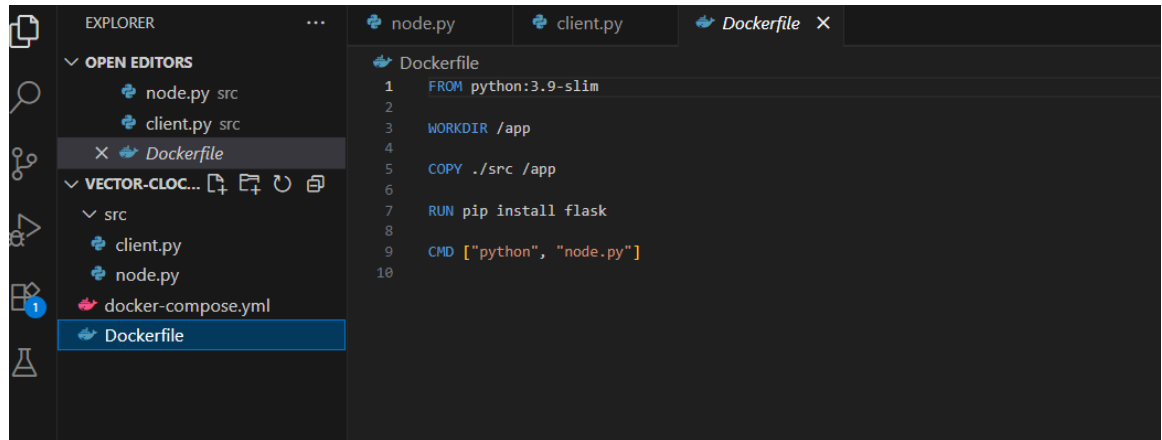
```
66
67 @app.route('/get', methods=['GET'])
68 def get():
69     key = request.args.get('key')
70     value = store.get(key, None)
71     print(f"[{node_id}] get {key}={value}")
72     return {'value': value}
73
74 @app.route('/replicate', methods=['POST'])
75 def replicate():
76     return put()
77
78 # ----- Background Buffer Processing -----
79
80 def process_buffer():
81     global buffer
82     while True:
83         time.sleep(0.5)
84         for entry in buffer[:]: # Iterate over a copy
85             if vector_clock.is_causally_ready(entry['clock'], entry['sender']):
86                 store[entry['key']] = entry['value']
87                 vector_clock.update(entry['clock'])
88                 buffer.remove(entry)
89                 print(f"[{node_id}] Buffered write applied: {entry['key']}={entry['value']}")
90
91 # ----- Node Initialization -----
92
93 def start_node(my_id, node_list):
94     global node_id, all_nodes, vector_clock
95     node_id = my_id
96     all_nodes = node_list
97     vector_clock = VectorClock(node_id, all_nodes)
98
99     print(f"[{node_id}] Node started with clock: {vector_clock.clock}")
100     threading.Thread(target=process_buffer, daemon=True).start()
101     app.run(host="0.0.0.0", port=5000)
102
103 # ----- Entry Point -----
104
105 if __name__ == '__main__':
106     my_node_id = sys.argv[1] # e.g., "node1"
107     node_ids = sys.argv[2].split(',') # e.g., "node1,node2,node3"
108     start_node(my_node_id, node_ids)
109
```

- Docker-compose.yml

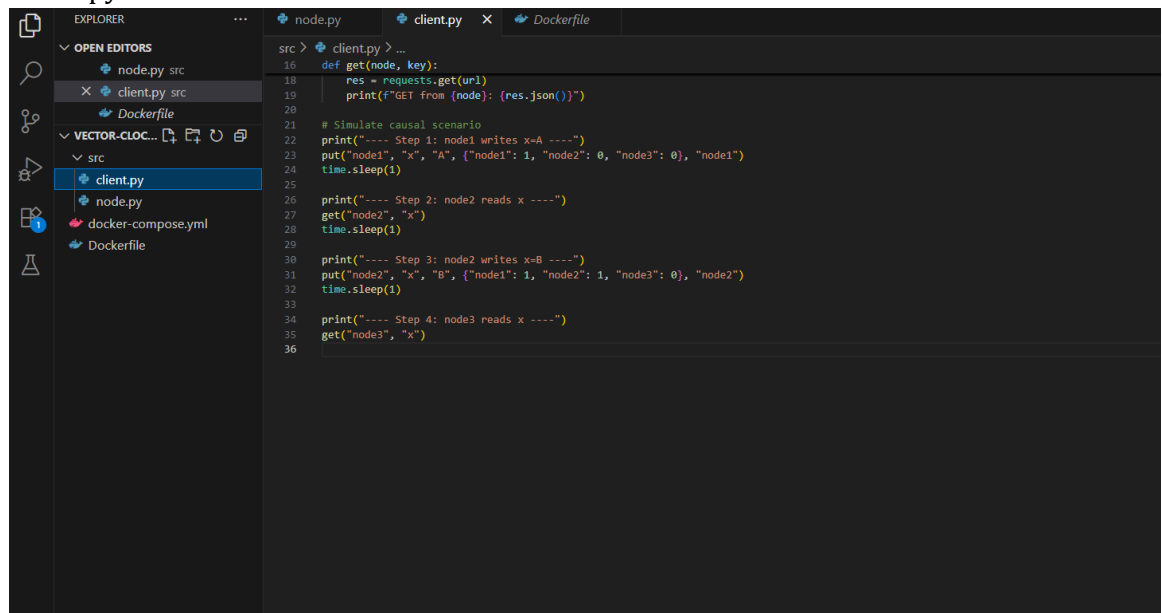


```
1 version: "3"
2 services:
3   node1:
4     build: .
5     ports:
6       - "5001:5000"
7     command: ["python", "node.py", "node1", "node1,node2,node3"]
8
9   node2:
10    build: .
11    ports:
12      - "5002:5000"
13    command: ["python", "node.py", "node2", "node1,node2,node3"]
14
15   node3:
16    build: .
17    ports:
18      - "5003:5000"
19    command: ["python", "node.py", "node3", "node1,node2,node3"]
20
```

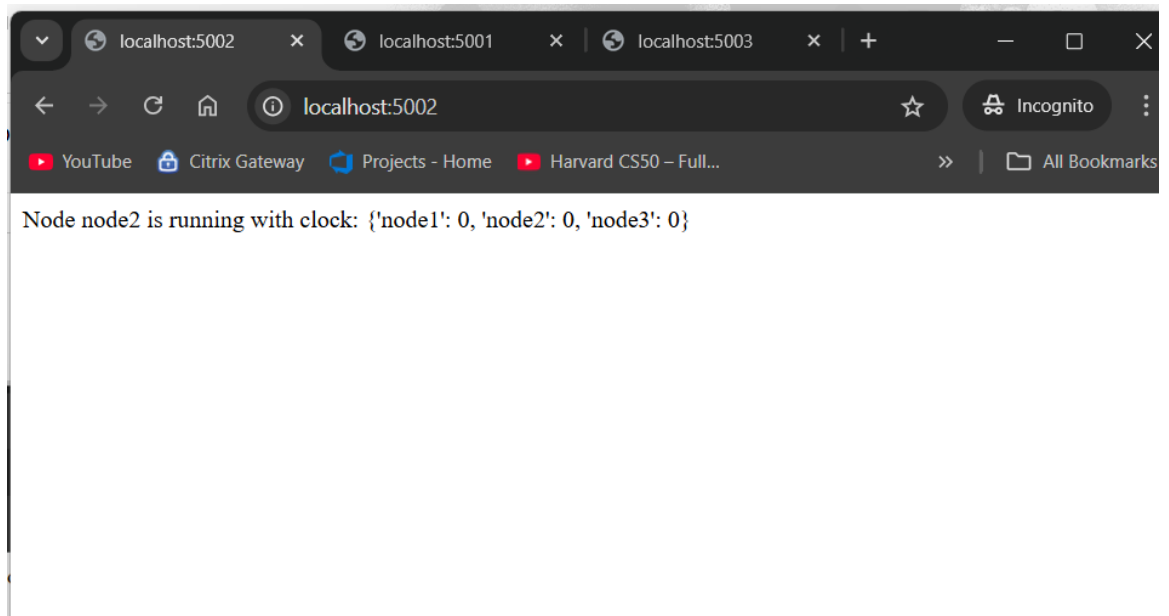
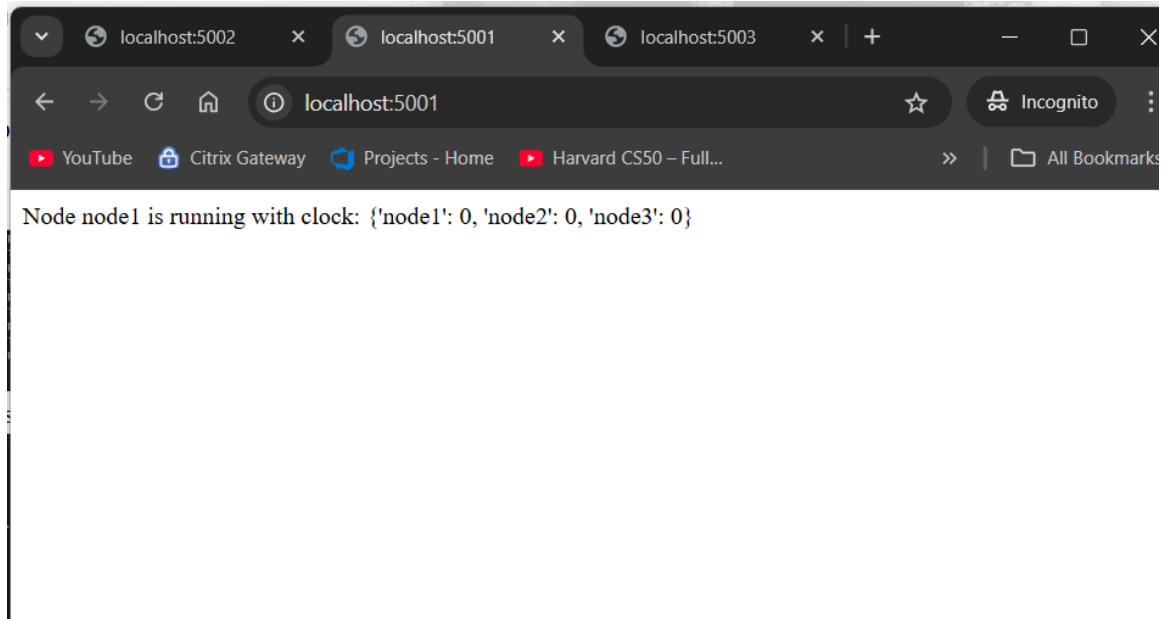
- Dockerfile

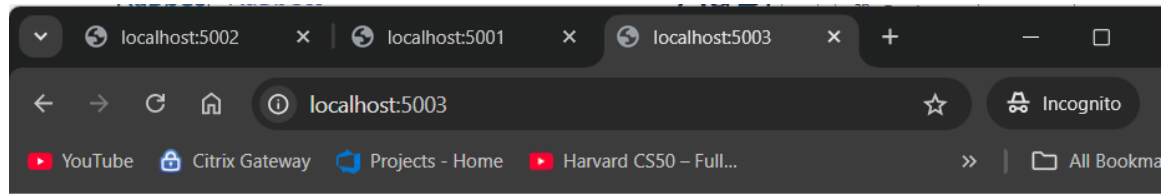


- Client.py



- browser response for node status





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

6. Testing and Results

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\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}
```

7. Video Demonstration

https://drive.google.com/file/d/1glBNEF7jOfEWErT3Ladfa_11Y24musbH/view?usp=drive_link

8. Conclusion

This project demonstrates the successful implementation of a **causally consistent distributed system** using **vector clocks**. All requirements are met:

- Vector Clock logic

Harshita Gupta (g24ai2017)

- Causal write propagation and buffering
- Flask APIs
- Containerized multi-node setup
- Scenario-based validation with a client script

End of Report