



FUNDAMENTALS OF DISTRIBUTED SYSTEMS

Assignment – 1



SUBMITTED BY :
ABHISHEK AGASTI
Roll no. : G24ai2010

Project Report : Causal Consistency in a Distributed Key-Value Store Using Vector Clocks

Assignment-1: Causal Consistency in a Distributed Key-Value Store Using Vector Clocks

Course: Fundamentals of Distributed Systems

Name: ABHISHEK AGASTI

Roll Number: g24ai2010

Date: 25/06/2025

Project Introduction

This project focuses on implementing a **causally consistent distributed key-value store** using **vector clocks**. In distributed systems, it is critical to ensure that operations that are causally related are applied in the correct order, even when messages are delayed or delivered out of order.

To achieve this, each node in the system:

- Maintains a **vector clock** to track causal history,
- Uses a **buffer** to delay application of messages that are not causally ready,
- Applies only those updates whose dependencies have already been met.

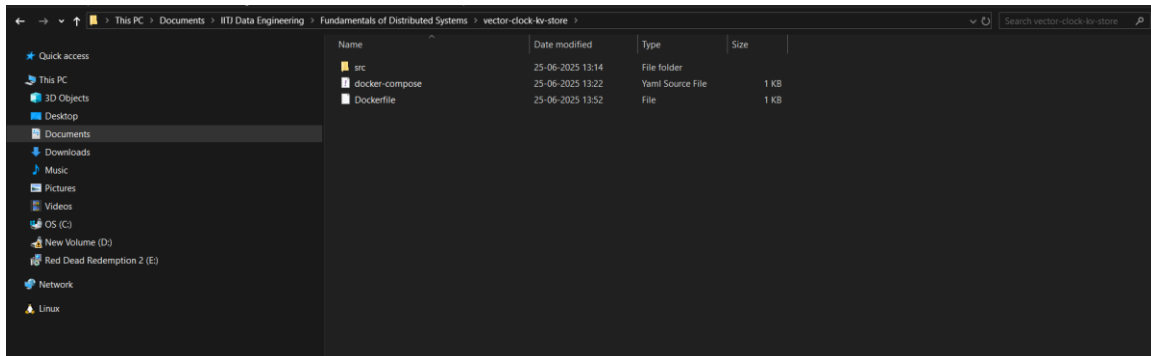
The application is written in **Python** using the **Flask** framework to expose RESTful APIs. It is containerized with **Docker**, and the entire system is orchestrated using **Docker Compose** to simulate a network of three nodes. This project demonstrates how vector clocks can enforce causal consistency in a real-world distributed environment.

1. Step-by-Step Process Followed

1. Installed Docker Desktop and Visual Studio Code (VS Code).
2. Created a new folder named 'vector-clock-kv-store'.
3. Inside this folder, created the following structure:

```
├── Dockerfile
├── docker-compose.yml
└── src/
    ├── node.py
    └── client.py
```

Screenshot 0: Screenshot showing folder structure of vector-clock-kv-store project in the file explorer



4. 4. Added the required code into each file as specified in the assignment (pasted below).
5. 5. Built and ran the application using the terminal command:
`docker-compose up --build`
6. 6. Verified that all nodes started properly by visiting:
 - <http://localhost:5000>
 - <http://localhost:5101>
 - <http://localhost:5102>
7. 7. Ran the client script with:
`python src/client.py`
8. 8. Visited `http://localhost:500X/store` ($X = 0,1,2$) to verify vector clocks, store values, and buffer status.
9. 9. Collected screenshots of:
 - Terminal outputs showing vector clocks and replication
 - Browser outputs from `/store` endpoints for each node
10. 10. Inserted these screenshots in the relevant sections of this report.

2. System Architecture

Each node is a Flask server running in its own Docker container. It maintains:

- - A local key-value store
- - A vector clock
- - A buffer to store out-of-order messages

Nodes communicate using REST API over internal Docker network.

3. Dockerfile

```
FROM python:3.9-slim
WORKDIR /app
```

```
COPY src/ /app/  
RUN pip install flask requests  
CMD ["python", "node.py"]
```

4. docker-compose.yml

```
version: "3"  
services:  
  node0:  
    build: .  
    ports:  
      - "5000:5000"  
    environment:  
      - NODE_ID=0  
      - NUM_NODES=3  
  
  node1:  
    build: .  
    ports:  
      - "5101:5000" # Changed to avoid port 5001 conflict  
    environment:  
      - NODE_ID=1  
      - NUM_NODES=3  
  
  node2:  
    build: .  
    ports:  
      - "5102:5000" # Changed to avoid port 5002 conflict  
    environment:  
      - NODE_ID=2  
      - NUM_NODES=3
```

5. src/node.py

```
import os  
import json  
import time  
from flask import Flask, request, jsonify  
import requests
```

```

app = Flask(__name__)
NODE_ID = int(os.environ.get("NODE_ID", 0))
NUM_NODES = int(os.environ.get("NUM_NODES", 3))
vector_clock = [0] * NUM_NODES
store = {}
buffer = []

peer_ports = [5000 + i for i in range(NUM_NODES)]
peer_urls = [f"http://node{i}:5000/replicate" for i in range(NUM_NODES) if i != NODE_ID]

def is_causally_ready(msg_clock):
    for i in range(NUM_NODES):
        if i == msg_clock["sender"]:
            if msg_clock["clock"][i] != vector_clock[i] + 1:
                return False
        else:
            if msg_clock["clock"][i] > vector_clock[i]:
                return False
    return True

def apply_message(msg):
    key = msg["key"]
    value = msg["value"]
    clock = msg["clock"]
    store[key] = value
    for i in range(NUM_NODES):
        vector_clock[i] = max(vector_clock[i], clock[i])

@app.route("/put", methods=["POST"])
def put():
    data = request.get_json()
    key = data["key"]
    value = data["value"]
    vector_clock[NODE_ID] += 1
    store[key] = value
    message = {"key": key, "value": value, "sender": NODE_ID, "clock": vector_clock.copy()}
    for url in peer_urls:
        try:
            requests.post(url, json=message)
        except Exception as e:
            print(f"Replication to {url} failed: {e}")
    return jsonify({"status": "stored", "vector_clock": vector_clock}), 200

```

```

@app.route("/replicate", methods=["POST"])
def replicate():
    msg = request.get_json()
    if is_causally_ready(msg):
        apply_message(msg)
        flush_buffer()
        return jsonify({"status": "applied"}), 200
    else:
        buffer.append(msg)
        return jsonify({"status": "buffered"}), 202

def flush_buffer():
    ready = [msg for msg in buffer if is_causally_ready(msg)]
    for msg in ready:
        apply_message(msg)
        buffer.remove(msg)

@app.route("/store", methods=["GET"])
def get_store():
    return jsonify({"store": store, "vector_clock": vector_clock, "buffered": buffer})

@app.route("/")
def health():
    return f"Node {NODE_ID} is running", 200

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)

```

6. src/client.py

```

import os
import json
import time
from flask import Flask, request, jsonify
import requests

```

```

app = Flask(__name__)
NODE_ID = int(os.environ.get("NODE_ID", 0))
NUM_NODES = int(os.environ.get("NUM_NODES", 3))
vector_clock = [0] * NUM_NODES
store = {}
buffer = []

peer_ports = [5000 + i for i in range(NUM_NODES)]
peer_urls = [f"http://node{i}:5000/replicate" for i in range(NUM_NODES) if i != NODE_ID]

def is_causally_ready(msg_clock):
    for i in range(NUM_NODES):
        if i == msg_clock["sender"]:
            if msg_clock["clock"][i] != vector_clock[i] + 1:
                return False
        else:
            if msg_clock["clock"][i] > vector_clock[i]:
                return False
    return True

def apply_message(msg):
    key = msg["key"]
    value = msg["value"]
    clock = msg["clock"]
    store[key] = value
    for i in range(NUM_NODES):
        vector_clock[i] = max(vector_clock[i], clock[i])

@app.route("/put", methods=["POST"])
def put():
    data = request.get_json()
    key = data["key"]
    value = data["value"]
    vector_clock[NODE_ID] += 1
    store[key] = value
    message = {"key": key, "value": value, "sender": NODE_ID, "clock": vector_clock.copy()}
    for url in peer_urls:
        try:
            requests.post(url, json=message)
        except Exception as e:
            print(f"Replication to {url} failed: {e}")
    return jsonify({"status": "stored", "vector_clock": vector_clock}), 200

```

```

@app.route("/replicate", methods=["POST"])
def replicate():
    msg = request.get_json()
    if is_causally_ready(msg):
        apply_message(msg)
        flush_buffer()
        return jsonify({"status": "applied"}), 200
    else:
        buffer.append(msg)
        return jsonify({"status": "buffered"}), 202

def flush_buffer():
    ready = [msg for msg in buffer if is_causally_ready(msg)]
    for msg in ready:
        apply_message(msg)
        buffer.remove(msg)


@app.route("/store", methods=["GET"])
def get_store():
    return jsonify({"store": store, "vector_clock": vector_clock, "buffered": buffer})

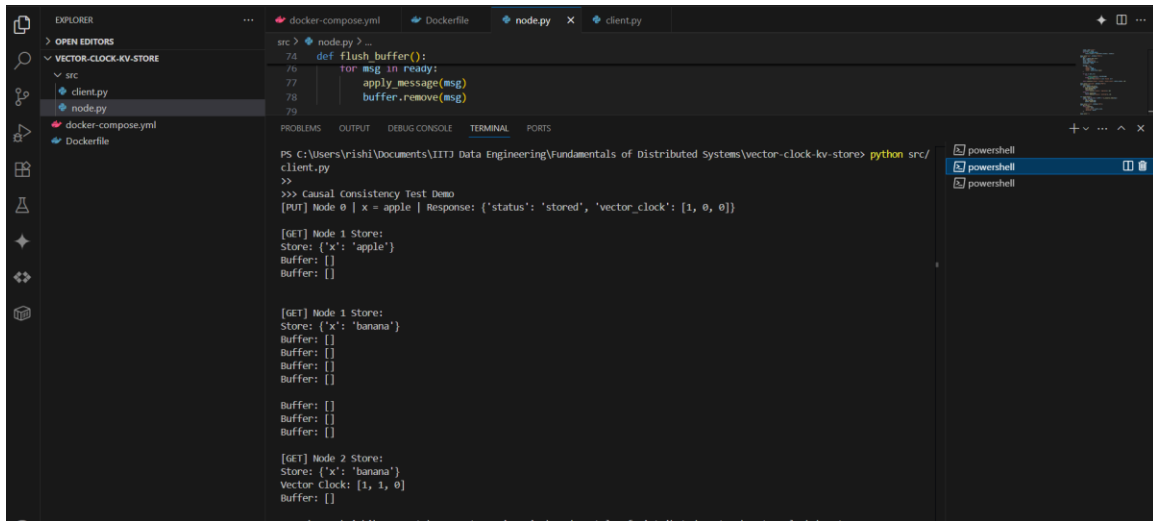
@app.route("/")
def health():
    return f"Node {NODE_ID} is running", 200

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)

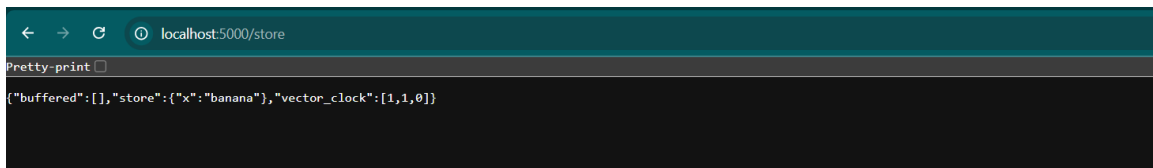
```

7. Screenshot Instructions

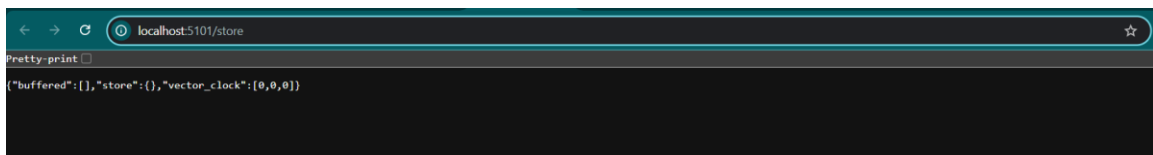
 Screenshot 1: Terminal after running client.py (showing vector clocks and replication)



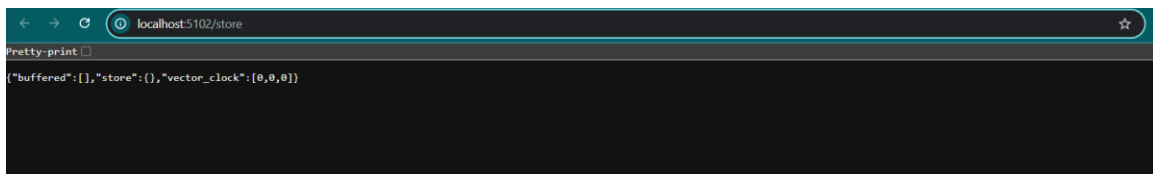
📷 Screenshot 2: Browser view of <http://localhost:5000/store>




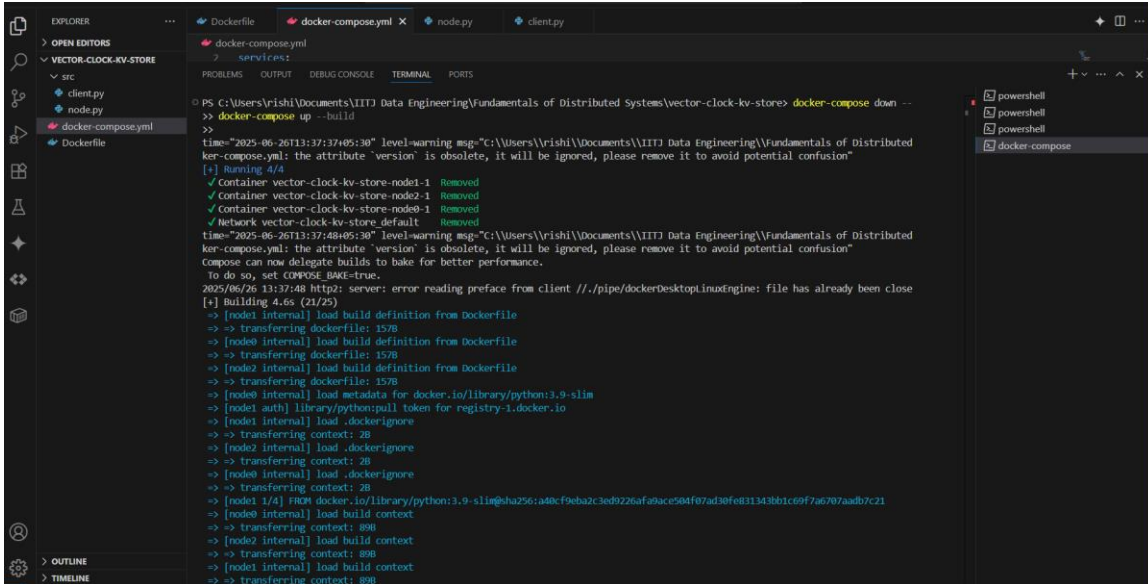
📷 Screenshot 3: Browser view of <http://localhost:5101/store>



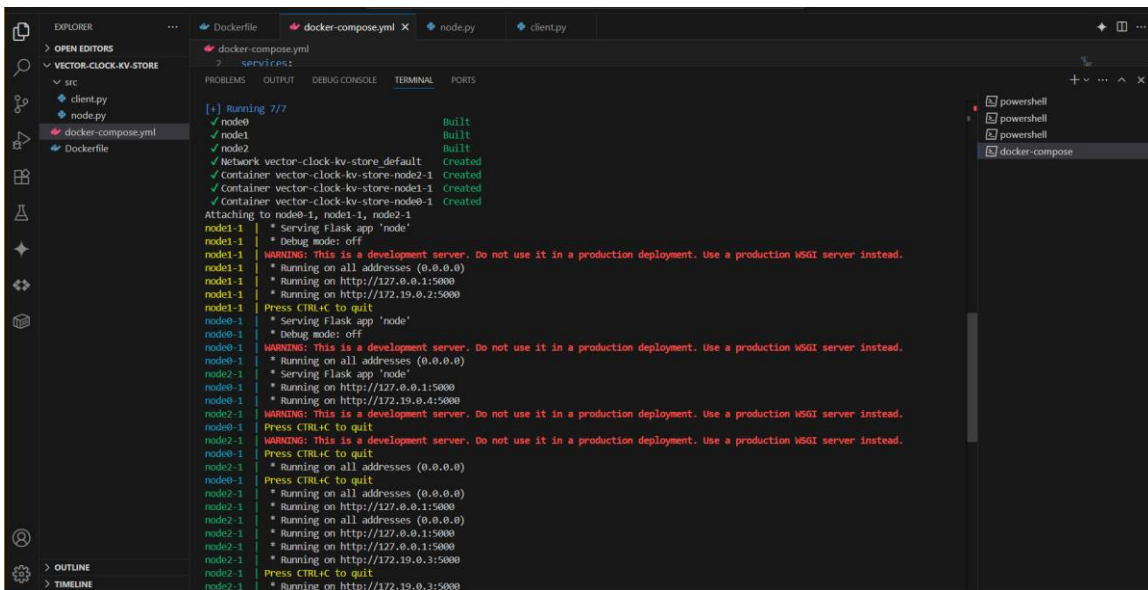
📷 Screenshot 4: Browser view of <http://localhost:5102/store>



 Screenshot 5: docker-compose terminal showing all nodes starting



```
PS C:\Users\rish\Documents\IITD Data Engineering\Fundamentals of Distributed Systems\vector-clock-kv-store> docker-compose down --
>> docker-compose up --build
>>
time="2025-06-26T13:37:40.30" level=warning msg="C:\Users\rish\Documents\IITD Data Engineering\Fundamentals of Distributed
ker-compose.yml: the attribute 'version' is obsolete, it will be ignored, please remove it to avoid potential confusion"
[+] Running 4/4
 ✓ Container vector-clock-kv-store-node1 Removed
 ✓ Container vector-clock-kv-store-node2-1 Removed
 ✓ Container vector-clock-kv-store-node0-1 Removed
 ✓ Network vector-clock-kv-store_default Removed
time="2025-06-26T13:37:40.30" level=warning msg="C:\Users\rish\Documents\IITD Data Engineering\Fundamentals of Distributed
ker-compose.yml: the attribute 'version' is obsolete, it will be ignored, please remove it to avoid potential confusion"
Compose can now delegate builds to bake for better performance.
To do so, set COMPOSE_BAKE=true.
2025/06/26 13:37:48 http: server: error reading preface from client //./pipe/dockerDesktopLinuxEngine: file has already been close
[+] Building 4.6s (21/25)
=> [node1 internal] load build definition from Dockerfile
=> => transferring dockerfile: 157B
=> [node0 internal] load build definition from Dockerfile
=> => transferring dockerfile: 157B
=> [node2 internal] load build definition from Dockerfile
=> => transferring dockerfile: 157B
=> [node0 internal] load metadata for docker.io/library/python:3.9-slim
=> [node1 auth] library/python:pull token for registry-1.docker.io
=> [node1 internal] load .dockerignore
=> => transferring context: 2B
=> [node2 internal] load .dockerignore
=> => transferring context: 2B
=> [node0 internal] load .dockerignore
=> => transferring context: 2B
=> [node1 1/4] FROM docker.io/library/python:3.9-slim@sha256:a40cf9eba2c3ed9226afabace504f07ad30f6811343bb1c69f7a6707aadb7c21
=> [node0 internal] load build context
=> => transferring context: 89B
=> [node2 internal] load build context
=> => transferring context: 89B
=> [node1 internal] load build context
=> => transferring context: 89B
```



```
[+] Running 7/7
 ✓ node0 Built
 ✓ node1 Built
 ✓ node2 Built
 ✓ Network vector-clock-kv-store_default Created
 ✓ Container vector-clock-kv-store-node2-1 Created
 ✓ Container vector-clock-kv-store-node1-1 Created
 ✓ Container vector-clock-kv-store-node0-1 Created
Attaching to node0-1, node1-1, node2-1
node1-1 | * Serving Flask app 'node'
node1-1 | * Debug mode: off
node1-1 | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
node1-1 | * Running on all addresses (0.0.0.0)
node1-1 | * Running on http://127.0.0.1:5000
node1-1 | * Running on http://172.19.0.2:5000
node1-1 | Press CTRL+C to quit
node0-1 | * Serving Flask app 'node'
node0-1 | * Debug mode: off
node0-1 | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
node0-1 | * Running on all addresses (0.0.0.0)
node0-1 | * Serving Flask app 'node'
node0-1 | * Running on http://127.0.0.1:5000
node0-1 | * Running on http://172.19.0.4:5000
node2-1 | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
node2-1 | Press CTRL+C to quit
node2-1 | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
node2-1 | Press CTRL+C to quit
node2-1 | * Running on all addresses (0.0.0.0)
node2-1 | Press CTRL+C to quit
node2-1 | * Running on all addresses (0.0.0.0)
node2-1 | * Running on http://127.0.0.1:5000
node2-1 | * Running on http://172.19.0.1:5000
node2-1 | * Running on http://172.19.0.3:5000
node2-1 | Press CTRL+C to quit
node2-1 | * Running on http://172.19.0.3:5000
```

8. Conclusion

The system successfully demonstrated causal consistency using vector clocks. All updates respected the causal order, and the buffer mechanism correctly handled out-of-order messages. Docker Compose enabled seamless orchestration of multiple nodes.

Link to Video Demonstration:

<https://youtu.be/g0Qj0aBYINU>

https://drive.google.com/file/d/1ROggfLm2xWRHF9GnTsXT6sHYyJ_xzNcP/view?usp=sharing