

Causally Consistent Key-Value Store using Vector Clocks

Project Report

Prepared by: Anjali Garg (g24ai2104)

Date: 23-06-2025

Contents

1 Objective.....	2
2 Architecture Overview.....	2
2.1 System Components.....	2
2.2 Key Features.....	3
3 Implementation Details.....	3
3.1 Vector Clock Logic.....	3
3.2 Node Operations.....	3
3.3 Causal Consistency Rules.....	4
4 Test Scenario & Verification.....	4
4.1 Causal Consistency Test.....	4
4.2 Out of Order Delivery Test.....	5
5. Conclusion.....	6

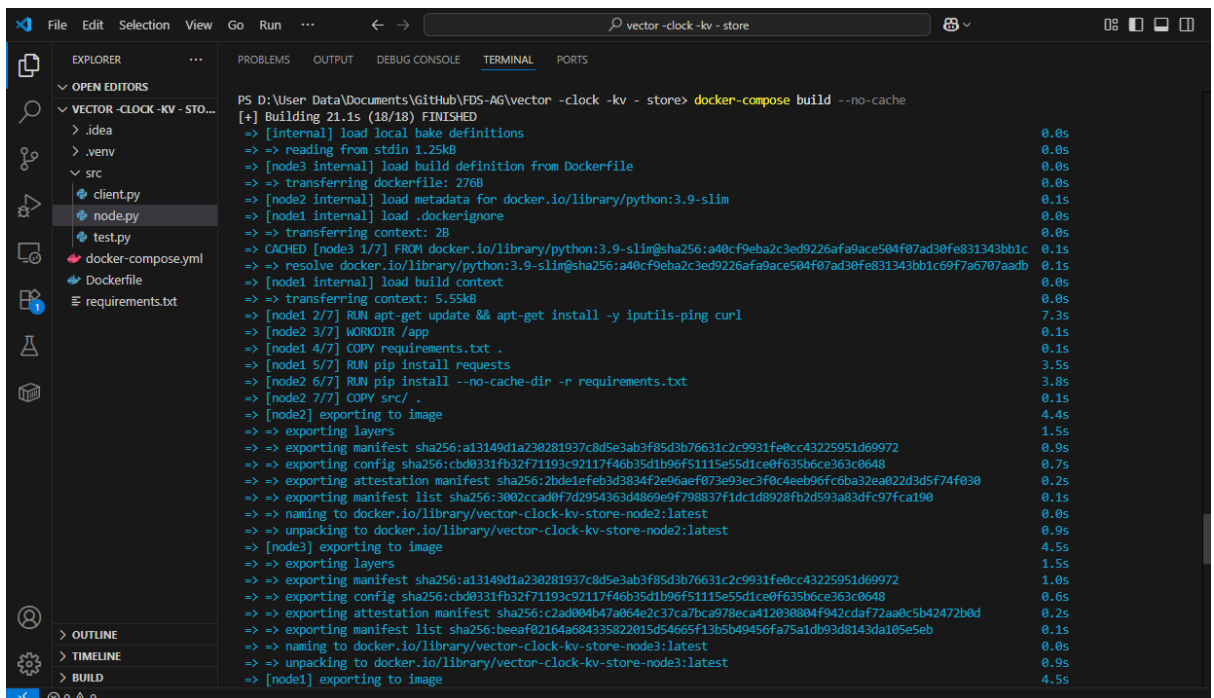
1 Objective

To implement a distributed, multi-node key-value store that ensures **causal consistency** by using **Vector Clocks** to track the causal relationship between events across nodes.

2 Architecture Overview

2.1 System Components

- **Nodes:** Independent Flask-based services representing key-value store instances. Each node maintains:
 - A **local vector clock**
 - A **local key-value store**
 - A **replication buffer** for causality-enforcing message delivery
- **Client:** A Python script to simulate operations and validate causal consistency across nodes
- **Docker Compose:** Used to containerize and orchestrate the three-node system



```
PS D:\User Data\Documents\GitHub\FDS-AG\vector -clock -kv - store> docker-compose build --no-cache
[+] Building 21.1s (18/18) FINISHED
=> [internal] load local bake definitions 0.0s
=> => reading from stdin 1.25KB 0.0s
=> [node3 internal] load build definition from Dockerfile 0.0s
=> => transferring dockerfile: 276B 0.0s
=> [node2 internal] load metadata for docker.io/library/python:3.9-slim 0.1s
=> [node1 internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> CACHED [node3 1/7] FROM docker.io/library/python:3.9-slim@sha256:a40cf9eba2c3ed9226afa9ace504f07ad30fe831343bb1c 0.1s
=> => resolve docker.io/library/python:3.9-slim@sha256:a40cf9eba2c3ed9226afa9ace504f07ad30fe831343bb1c69f7a6707aadb 0.1s
=> [node1 internal] load build context 0.0s
=> => transferring context: 5.55kB 0.0s
=> [node1 2/7] RUN apt-get update && apt-get install -y iputils-ping curl 7.3s
=> [node2 3/7] WORKDIR /app 0.1s
=> [node1 4/7] COPY requirements.txt . 0.1s
=> [node1 5/7] RUN pip install requests 3.5s
=> [node2 6/7] RUN pip install --no-cache-dir -r requirements.txt 3.8s
=> [node2 7/7] COPY src/ . 0.1s
=> [node2] exporting to image 4.4s
=> => exporting layers 1.5s
=> => exporting manifest sha256:a13149d1a230281937c8d5e3ab3f85d3b76631c2c9931fe0cc43225951d69972 0.9s
=> => exporting config sha256:cbd0331fb32f71193c92117f46b35d1b96f51115e55d1ce0f635b6ce363c0648 0.7s
=> => exporting attestation manifest sha256:2bde1efeb3d3834f2e96aef073e93ec3f0c4eeb96fc6ba32ea022d3d5f74f030 0.2s
=> => exporting manifest list sha256:3002ccad0f7d2954363d4869e9f798837f1dc1d8928fb2d593a83dfc97fca190 0.1s
=> => naming to docker.io/library/vector-clock-kv-store-node2:latest 0.0s
=> => unpacking to docker.io/library/vector-clock-kv-store-node2:latest 0.9s
=> [node3] exporting to image 4.5s
=> => exporting layers 1.5s
=> => exporting manifest sha256:a13149d1a230281937c8d5e3ab3f85d3b76631c2c9931fe0cc43225951d69972 1.0s
=> => exporting config sha256:cbd0331fb32f71193c92117f46b35d1b96f51115e55d1ce0f635b6ce363c0648 0.6s
=> => exporting attestation manifest sha256:c2ad004b47a064e2c37ca7bca978eca412030804f942cdf72aa0c5b42472b0d 0.2s
=> => exporting manifest list sha256:beeaf02164a684335822015d54665f13b5b49456fa75a1db93d8143da105e5eb 0.1s
=> => naming to docker.io/library/vector-clock-kv-store-node3:latest 0.0s
=> => unpacking to docker.io/library/vector-clock-kv-store-node3:latest 0.9s
=> [node1] exporting to image 4.5s
```

- **3 nodes created**

```

=> => exporting layers 1.5s
=> => exporting manifest sha256:a13149d1a230281937c8d5e3ab3f85d3b76631c2c9931fe0cc43225951d69972 0.9s
=> => exporting config sha256:cbd0331fb32f71193c92117f46b35d1b06f51115e55d1ce0f635b6ce363c0648 0.7s
=> => exporting attestation manifest sha256:b47ca13385b968d6b33d3cbe3be06ad4352d30888e848fd6eeab6680dc9c090e 0.2s
=> => exporting manifest list sha256:b993ec40a32e4893cb58053e7354b590fc5551f10da0a7ef11e30ba11614f3bd 0.1s
=> => naming to docker.io/library/vector-clock-kv-store-node1:latest 0.0s
=> => unpacking to docker.io/library/vector-clock-kv-store-node1:latest 0.9s
=> [node2] resolving provenance for metadata file 0.1s
=> [node1] resolving provenance for metadata file 0.1s
=> [node3] resolving provenance for metadata file 0.0s
[+] Building 3/3
✓ node3 Built 0.0s
✓ node1 Built 0.0s
✓ node2 Built 0.0s
PS D:\User Data\Documents\GitHub\FDS-AG\vector -clock -kv - store> docker-compose up -d
[+] Running 4/4
✓ Network kvstore_network Created 0.0s
✓ Container node2 Started 0.7s
✓ Container node3 Started 0.9s
✓ Container node1 Started 0.9s
PS D:\User Data\Documents\GitHub\FDS-AG\vector -clock -kv - store> docker-compose ps

```

NAME	IMAGE	COMMAND	SERVICE	CREATED	STATUS	PORTS
node1	vector-clock-kv-store-node1	"python node.py"	node1	4 seconds ago	Up 3 seconds	0.0.0.0:5001->5000/tcp, [::]:5001->5000/tcp
node2	vector-clock-kv-store-node2	"python node.py"	node2	4 seconds ago	Up 3 seconds	0.0.0.0:5002->5000/tcp, [::]:5002->5000/tcp
node3	vector-clock-kv-store-node3	"python node.py"	node3	4 seconds ago	Up 3 seconds	0.0.0.0:5003->5000/tcp, [::]:5003->5000/tcp

2.2 Key Features

- Nodes replicate updates to all other nodes using HTTP POST.
- Vector clocks are sent along with every replicated write.
- Each node uses the **Causal Delivery Rule** to determine whether an update can be immediately applied or should be buffered.

Vector Clock Rules:

- On a local write: increment the node's clock.
- On receive: compare vector clock to decide if causal dependencies are met.
- Buffer updates if causal dependencies are not yet satisfied.

3 Implementation Details

3.1 Vector Clock Logic

- Each node tracks a dictionary-based vector clock (e.g., {"node1": 1, "node2": 0, "node3": 0}).
 - **On local write:** Increment own clock
 - **On receiving a write:** Merge clocks and check causality

3.2 Node Operations

- /write: Handles client writes (increments clock, replicates)

- /receive: Processes replicated writes (checks causality, buffers if needed)
- /read: Returns key-value pair with latest clock

3.3 Causal Consistency Rules

- A write is applied only if:
 - All preceding operations (per vector clock) have been processed
 - Otherwise, it is buffered and retried later

4 Test Scenario & Verification

4.1 Causal Consistency Test

Using client.py, the following scenario is executed:

1. Write x=5 to node1

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
node1 | * Running on http://172.20.0.3:5000
node1 | Press CTRL+C to quit
node1 | 172.20.0.1 - - [23/Jun/2025 07:57:29] "GET /health HTTP/1.1" 200 -
node1 |
node1 | [node1] Received write request
node1 | [node1] Write: x=5, clock={'node1': 1, 'node2': 0, 'node3': 0}
node1 | [node1] Replicating to http://node2:5000/receive key=x, value=5
node1 | 172.20.0.1 - - [23/Jun/2025 07:57:29] "POST /write HTTP/1.1" 200 -
node1 | [node1] Replicating to http://node3:5000/receive key=x, value=5
node1 |
node1 | 172.20.0.2 - - [23/Jun/2025 07:57:30] "POST /receive HTTP/1.1" 200 -
node1 | [node1] Received replication: {'key': 'x', 'value': 10, 'sender': 'node2', 'timestamp': {'node1': 1, 'node2': 1, 'node3': 0}}
node1 | [node1] Applied replication: x=10, clock={'node1': 1, 'node2': 1, 'node3': 0}
PS D:\User Data\Documents\GitHub\FDS-AG\vector -clock -kv - store>

```

2. Read x from node2, capturing its vector clock

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
node2 | 172.20.0.1 - - [23/Jun/2025 07:57:29] "GET /health HTTP/1.1" 200 -
node2 |
node2 | [node2] Received replication: {'key': 'x', 'value': 5, 'sender': 'node1', 'timestamp': {'node1': 1, 'node2': 0, 'node3': 0}}
node2 | [node2] Applied replication: x=5, clock={'node1': 1, 'node2': 0, 'node3': 0}
node2 | 172.20.0.3 - - [23/Jun/2025 07:57:29] "POST /receive HTTP/1.1" 200 -
node2 | 172.20.0.1 - - [23/Jun/2025 07:57:30] "GET /read?key=x HTTP/1.1" 200 -
node2 |

```

3. Update x=10 on node2, using the clock from step 2 as causal context

```

node2 | [node2] Received write request
node2 | [node2] Merged context from client: {'node1': 1, 'node2': 0, 'node3': 0}
node2 | [node2] Write: x=10, clock={'node1': 1, 'node2': 1, 'node3': 0}
node2 | [node2] Replicating to http://node1:5000/receive key=x, value=10
node2 | 172.20.0.1 - - [23/Jun/2025 07:57:30] "POST /write HTTP/1.1" 200 -
node2 | [node2] Replicating to http://node3:5000/receive key=x, value=10
PS D:\User Data\Documents\GitHub\FDS-AG\vector -clock -kv - store>

```

4. Read from node3 to validate that x=10 is seen only after x=5 is processed

```
node3 | [node3] Received replication: {'key': 'x', 'value': 5, 'sender': 'node1', 'timestamp': {'node1': 1, 'node2': 0, 'node3': 0}}
node3 | [node3] Applied replication: x=5, clock={'node1': 1, 'node2': 0, 'node3': 0}
node3 | 172.20.0.3 - - [23/Jun/2025 07:57:29] "POST /receive HTTP/1.1" 200 -
node3 | [node3] Received replication: {'key': 'x', 'value': 10, 'sender': 'node2', 'timestamp': {'node1': 1, 'node2': 1, 'node3': 0}}
node3 | [node3] Applied replication: x=10, clock={'node1': 1, 'node2': 1, 'node3': 0}
node3 | 172.20.0.2 - - [23/Jun/2025 07:57:30] "POST /receive HTTP/1.1" 200 -
node3 | 172.20.0.1 - - [23/Jun/2025 07:57:31] "GET /read?key=x HTTP/1.1" 200 -
PS D:\User Data\Documents\GitHub\FDS-AG\vector -clock -kv - store>
```

4.2 Out of Order Delivery Test

1. x=100 is written to node1 and artificially delayed in delivery to node3
2. Meanwhile, x=200 is written to node2 using the vector clock of the delayed message
3. node3 receives x=200 first, buffers it
4. Later, x=100 arrives, allowing x=200 to be applied correctly

```
20 node3 | [node3] Received replication: {'key': 'x', 'value': 200, 'sender': 'node2', 'timestamp': {'node1': 2, 'node2': 1, 'node3': 0}}
21 node3 | [node3] Cannot deliver yet, buffering...
22 node3 | 172.20.0.3 - - [23/Jun/2025 08:28:22] "POST /receive HTTP/1.1" 200 -
23 node3 |
24 node3 | [node3] Received replication: {'key': 'x', 'value': 100, 'sender': 'node1', 'timestamp': {'node1': 2, 'node2': 1, 'node3': 0}}
25 node3 | [node3] Applied replication: x=100, clock={'node1': 2, 'node2': 1, 'node3': 0}
26 node3 | 172.20.0.2 - - [23/Jun/2025 08:28:24] "POST /receive HTTP/1.1" 200 -
27 node3 | [node3] Applied replication: x=200, clock={'node1': 2, 'node2': 2, 'node3': 0}
28 node3 | 172.20.0.1 - - [23/Jun/2025 08:28:25] "GET /read?key=x HTTP/1.1" 200 -
29
```

```
=== Testing Out-of-Order Delivery Handling ===
Writing x=100 to node1...
Write successful. Clock: {'node1': 2, 'node2': 1, 'node3': 0}
Reading from node2...
Read value: 100 | Clock: {'node1': 2, 'node2': 1, 'node3': 0}
Updating to x=200 at node2...
Update successful. Clock: {'node1': 2, 'node2': 2, 'node3': 0}
Final value at node3: 200 | Clock: {'node1': 2, 'node2': 2, 'node3': 0}
✓ Out-of-order delivery handled correctly
```

```
🔥 All tests passed!
PS D:\User Data\Documents\GitHub\FDS-AG\vector -clock -kv - store>
```

Console Output

```
PS D:\User Data\Documents\GitHub\FDS-AG\vector -clock -kv - store> py src/client.py
Checking node availability...
✓ node1 ready | Clock: {'node1': 0, 'node2': 0, 'node3': 0}
✓ node2 ready | Clock: {'node1': 0, 'node2': 0, 'node3': 0}
✓ node3 ready | Clock: {'node1': 0, 'node2': 0, 'node3': 0}

=== Testing Causal Consistency ===
Writing x=5 to node1...
Write successful. Clock: {'node1': 1, 'node2': 0, 'node3': 0}
Reading from node2...
Read value: 5 | Clock: {'node1': 1, 'node2': 0, 'node3': 0}
Updating to x=10 at node2...
Update successful. Clock: {'node1': 1, 'node2': 1, 'node3': 0}
Final value at node3: 10 | Clock: {'node1': 1, 'node2': 1, 'node3': 0}
✓ Causal consistency verified

=== Testing Out-of-Order Delivery Handling ===
Writing x=100 to node1...
Write successful. Clock: {'node1': 2, 'node2': 1, 'node3': 0}
Reading from node2...
Read value: 100 | Clock: {'node1': 2, 'node2': 1, 'node3': 0}
Updating to x=200 at node2...
Update successful. Clock: {'node1': 2, 'node2': 2, 'node3': 0}
Final value at node3: 200 | Clock: {'node1': 2, 'node2': 2, 'node3': 0}
✓ Out-of-order delivery handled correctly

🔥 All tests passed!
PS D:\User Data\Documents\GitHub\FDS-AG\vector -clock -kv - store>
```

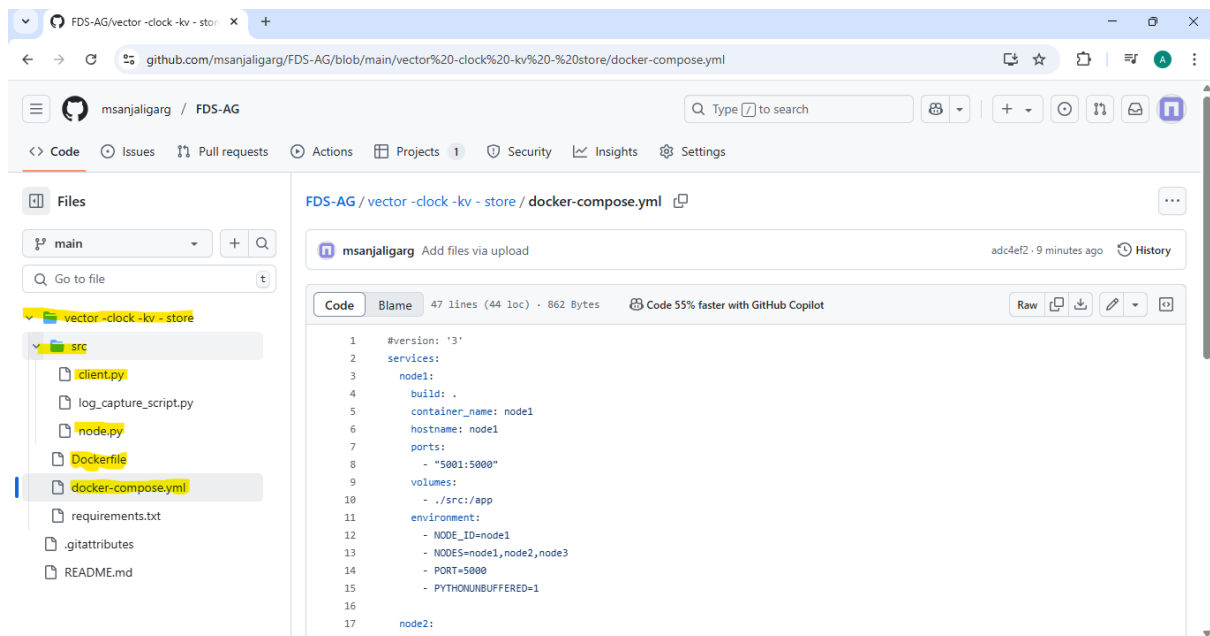
5. Conclusion

The project successfully implements causal consistency using vector clocks. Logs and output prove that:

- Nodes do not apply dependent writes until prior causal operations are received.
- Vector clocks are correctly merged and incremented.
- Buffered messages are correctly applied once causal constraints are met.

A demo [video](#) captures the architecture, operations, and both test cases

Folder structure in [Github](#)



The screenshot shows a GitHub repository for 'FDS-AG' by user 'msanjaligarg'. The left sidebar displays the file structure, with the 'vector-clock-kv-store' directory expanded, showing files like 'client.py', 'log_capture_script.py', 'node.py', 'Dockerfile', 'docker-compose.yml', 'requirements.txt', '.gitattributes', and 'README.md'. The main area shows the 'docker-compose.yml' file content, which is a Docker Compose configuration for a service named 'node1'.

```
1 #version: '3'
2 services:
3   node1:
4     build: .
5     container_name: node1
6     hostname: node1
7     ports:
8       - "5001:5000"
9     volumes:
10      - ./src:/app
11     environment:
12       - NODE_ID=node1
13       - NODES=node1,node2,node3
14       - PORT=5000
15       - PYTHONUNBUFFERED=1
16
17   node2:
```