

Objective

The objective of this project is to implement a distributed key-value store that maintains causal consistency using Vector Clocks. Unlike Lamport clocks, vector clocks help identify and preserve causal relationships between events in a distributed system.

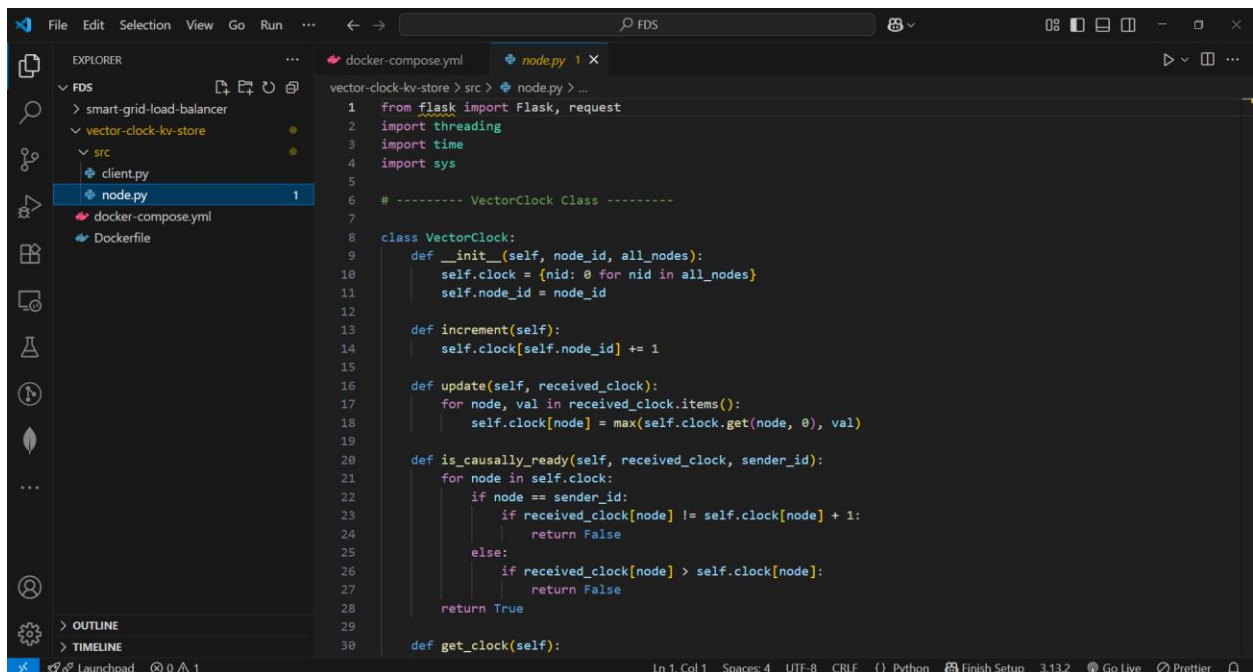
System Architecture

The system includes:

- **Three Flask-based Python nodes**
- **One client** to simulate PUT and GET operations
- **Docker and Docker Compose** for deployment and networking

Vector Clock Logic

- Each node maintains a vector clock with entries for all nodes.
- **Increment:** On every local write operation.
- **Update:** On receiving a vector clock from another node.
- **Causally Ready:** A write is only applied if:
 - For all $i \neq \text{sender}$, $\text{msg_clock}[i] \leq \text{local_clock}[i]$
 - And $\text{msg_clock}[\text{sender}] == \text{local_clock}[\text{sender}] + 1$

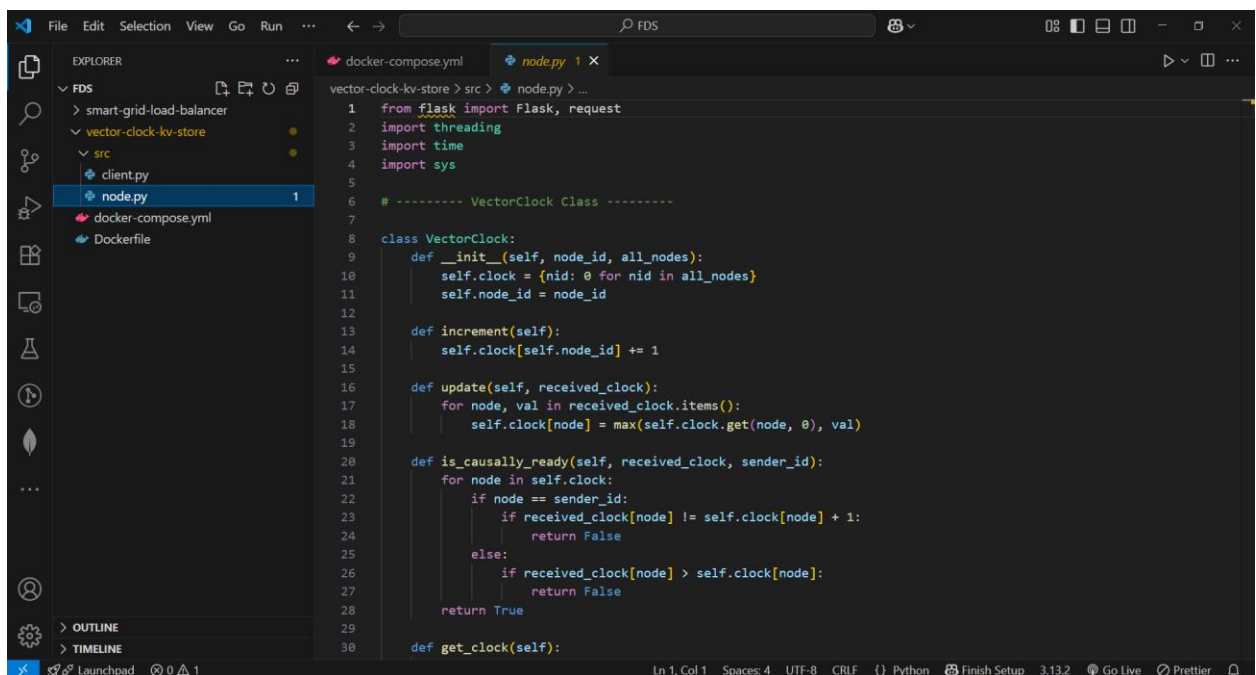


```
1 from flask import Flask, request
2 import threading
3 import time
4 import sys
5
6 # ----- VectorClock Class -----
7
8 class VectorClock:
9     def __init__(self, node_id, all_nodes):
10         self.clock = {nid: 0 for nid in all_nodes}
11         self.node_id = node_id
12
13     def increment(self):
14         self.clock[self.node_id] += 1
15
16     def update(self, received_clock):
17         for node, val in received_clock.items():
18             self.clock[node] = max(self.clock.get(node, 0), val)
19
20     def is_causally_ready(self, received_clock, sender_id):
21         for node in self.clock:
22             if node == sender_id:
23                 if received_clock[node] != self.clock[node] + 1:
24                     return False
25             else:
26                 if received_clock[node] > self.clock[node]:
27                     return False
28         return True
29
30     def get_clock(self):
```

Node Design (node.py)

Each node:

- Has a Flask server exposing:
 - GET /get?key=x
 - POST /replicate to receive updates
- Uses the VectorClock class to check causal readiness
- Buffers undeliverable messages until dependencies are resolved



```
1 from flask import Flask, request
2 import threading
3 import time
4 import sys
5
6 # ----- VectorClock Class -----
7
8 class VectorClock:
9     def __init__(self, node_id, all_nodes):
10         self.clock = {nid: 0 for nid in all_nodes}
11         self.node_id = node_id
12
13     def increment(self):
14         self.clock[self.node_id] += 1
15
16     def update(self, received_clock):
17         for node, val in received_clock.items():
18             self.clock[node] = max(self.clock.get(node, 0), val)
19
20     def is_causally_ready(self, received_clock, sender_id):
21         for node in self.clock:
22             if node == sender_id:
23                 if received_clock[node] != self.clock[node] + 1:
24                     return False
25             else:
26                 if received_clock[node] > self.clock[node]:
27                     return False
28         return True
29
30     def get_clock(self):
```

Client Design (client.py)

The client simulates the following scenario:

1. Step 1: node1 writes x = A
2. Step 2: node2 reads x
3. Step 3: node2 writes x = B, which is causally dependent on the previous read
4. Step 4: Ensure all nodes reflect consistent updates

The screenshot shows a VS Code editor window with the Explorer sidebar on the left. The Explorer shows a project structure with folders 'FDS', 'smart-grid-load-balancer', and 'vector-clock-kv-store'. Inside 'vector-clock-kv-store', there is a 'src' folder containing 'client.py', 'node.py', 'docker-compose.yml', and 'Dockerfile'. The main editor area displays the contents of 'client.py'. The code defines a Flask client with endpoints for PUT and GET requests, simulating a causal scenario with three nodes. The status bar at the bottom indicates the file is at line 10, column 42, with 4 spaces, UTF-8 encoding, and CRLF line endings. The status bar also shows the Python interpreter is set to 'Finish Setup' and the file is 3,132 bytes.

```
vector-clock-kv-store > src > client.py > put
1 import requests
2 import time
3
4 nodes = {
5     "node1": "http://localhost:5001",
6     "node2": "http://localhost:5002",
7     "node3": "http://localhost:5003",
8 }
9
10 def put(node, key, value, clock, sender):
11     url = f"{nodes[node]}/replicate"
12     data = {"key": key, "value": value, "clock": clock, "sender": sender}
13     res = requests.post(url, json=data)
14     print(f"PUT to {node}: {res.json()}")
15
16 def get(node, key):
17     url = f"{nodes[node]}/get?key={key}" # [X] Corrected here
18     res = requests.get(url)
19     print(f"GET from {node}: {res.json()}")
20
21 # Simulate causal scenario
22 print("---- Step 1: node1 writes x=A ----")
23 put("node1", "x", "A", {"node1": 1, "node2": 0, "node3": 0}, "node1")
24 time.sleep(1)
25
26 print("---- Step 2: node2 reads x ----")
27 get("node2", "x")
28 time.sleep(1)
29
30 print("---- Step 3: node2 writes x=B ----")
```

Containerization

- Each node runs in a separate container.
- docker-compose.yml defines the services and ports.
- Dockerfile installs Flask and runs node.py

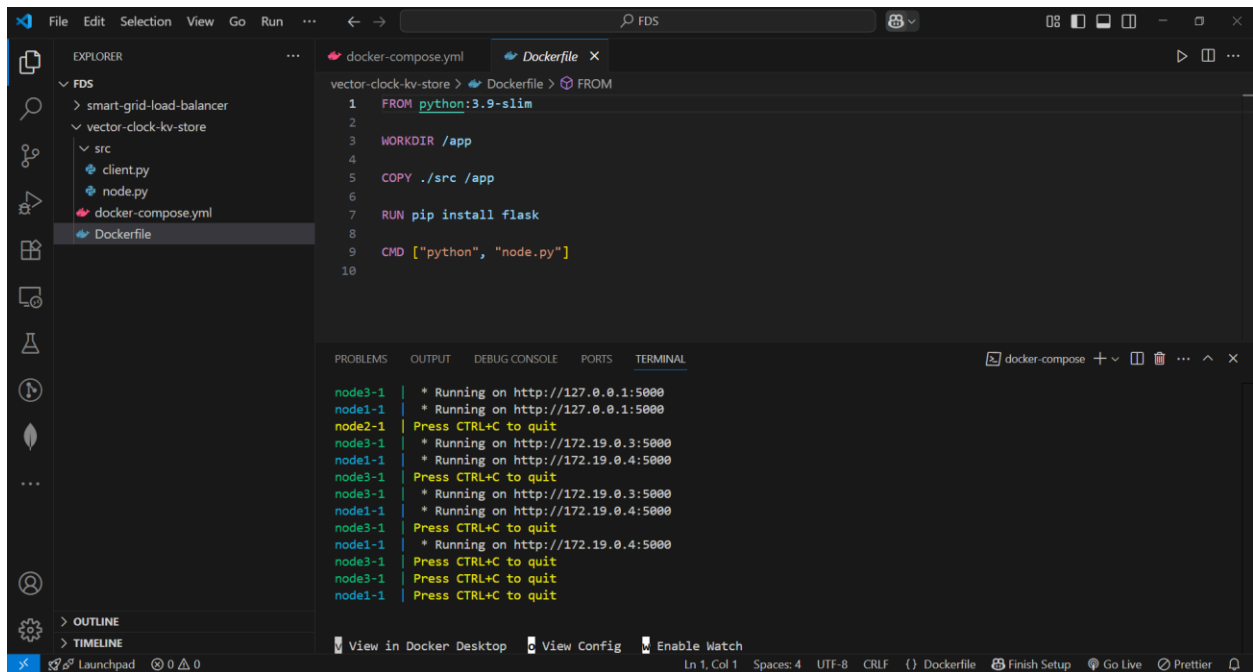
```
1 version: "3"
2 services:
3   node1:
4     build: .
5     ports:
6       - "5001:5000"
7     command: ["python", "node.py", "node1", "node1,node2,node3"]
8
9   node2:
10    build: .
11    ports:
12      - "5002:5000"
13    command: ["python", "node.py", "node2", "node1,node2,node3"]
14
15   node3: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

```
1 FROM python:3.9-slim
2
3 WORKDIR /app
4
5 COPY ./src /app
6
7 RUN pip install flask
8
9 CMD ["python", "node.py"]
10
```

Testing and Output

Terminal Output

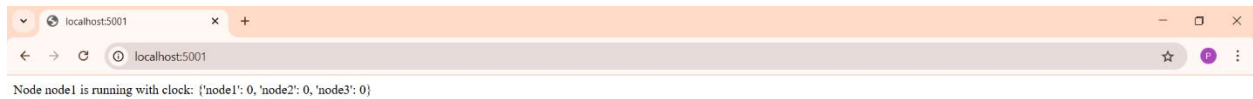
Logs confirm that all nodes start correctly and handle writes/reads as per causal rules.



Node Views

Each node displays its status and vector clock correctly.

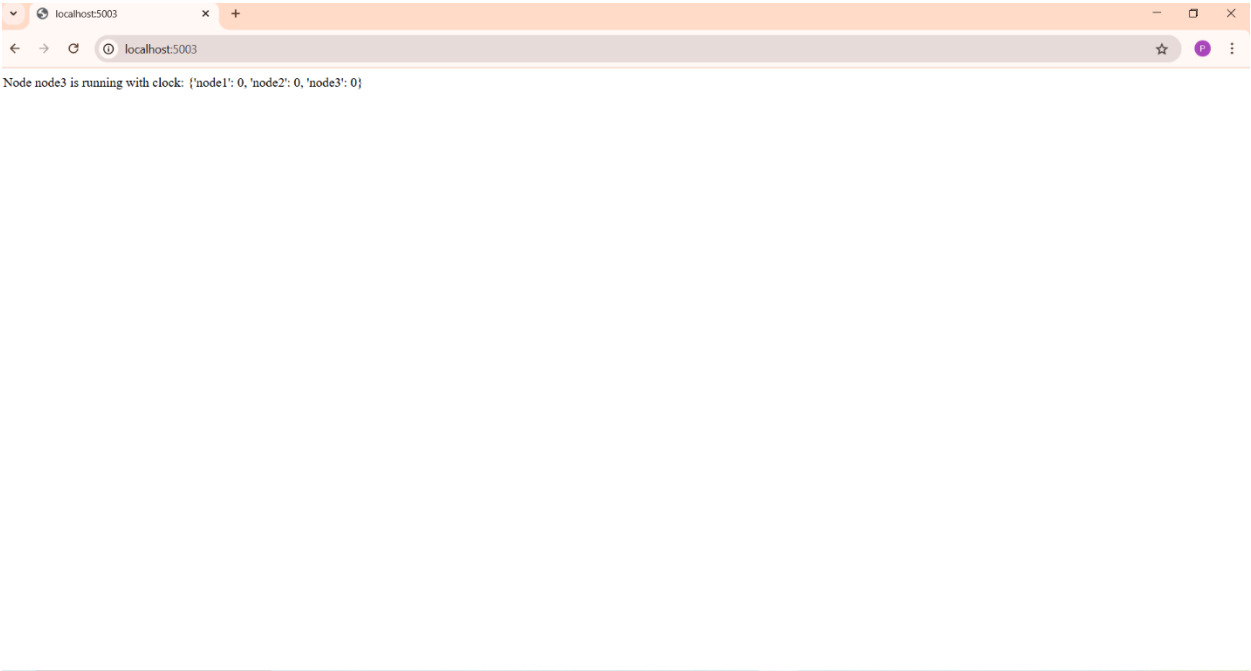
Node1



Node2



Node3



Conclusion

The system successfully maintains causal consistency using vector clocks. Buffered messages are only applied when their causal dependencies are met. Docker ensures the system is easily deployable and scalable.