## 1. Objective

The objective of this project is to build a distributed key-value store that guarantees **causal consistency** across three or more nodes. This is achieved using **vector clocks** to track and enforce the partial ordering of events in a distributed system. This goes beyond Lamport timestamps and enables nodes to detect and respect causal relationships among updates.

---

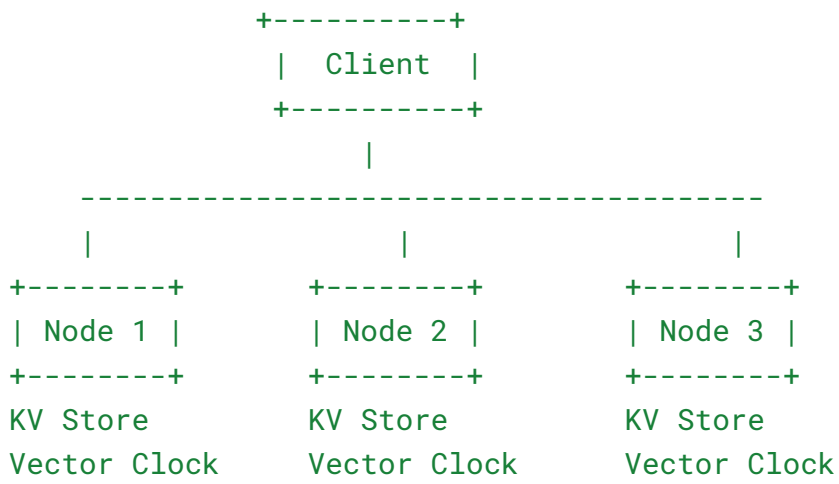
## 2. System Architecture

The system consists of the following components:

- **Nodes**: Each node stores its own local key-value data and maintains a vector clock to track causal history.
- **Client**: Sends read/write requests to nodes and drives the test scenarios.
- **Vector Clock**: A list-based logical clock that allows each node to compare the causality of events.
- **Message Buffering**: Writes that arrive before their dependencies are met are buffered until safe to apply.
- **Docker Orchestration**: All components run as Docker containers orchestrated using Docker Compose.

**Architecture Diagram**

lua
CopyEdit



---

## 3. Implementation Details

### 3.1 Node Logic (`node.py`)

- Each node maintains:
    - A local key-value store (`dict`)
    - Its own vector clock (`list`)
    - A buffer to hold incoming writes that cannot yet be applied
- On local write:
    - The node increments its own index in the vector clock.
    - The update is applied to its local store and propagated to other nodes.
- On receiving a remote write:
    - The node compares the incoming vector clock with its own.
    - If dependencies are met, the update is applied.
    - If not, the update is buffered for retry.

### 3.2 Client Script (`client.py`)

- The client supports:
    - `write <key> <value>` to any node
    - `read <key>` from any node
- Simulates realistic test scenarios including causal dependencies.

### 3.3 Vector Clock Comparison

Causal delivery is enforced using the following rule:
For node *i* to apply an incoming message *m* from node *j*:

- For all `k ≠ j`, `m.VC[k] ≤ local.VC[k]`
- `m.VC[j] == local.VC[j] + 1`

Messages that fail this check are buffered until their dependencies are met.

---

## 4. Docker and Deployment

- **Dockerfile**: Builds a container image for the Python node application.
- **docker-compose.yml**: Launches a three-node networked environment using Docker containers. Each node runs as an independent container and communicates via HTTP.

---

## 5. Causal Consistency Verification – Test Scenario

The following test demonstrates causal ordering enforcement:

**Commands Executed:**
bash
CopyEdit

```
python src/client.py http://localhost:5001 write x A
python src/client.py http://localhost:5002 write x B
python src/client.py http://localhost:5003 read x
```

**Output:**
Write to Node 1:

json
CopyEdit

```
{'key': 'x', 'node_id': 0, 'value': 'A', 'vector_clock': [1, 0, 0]}
```

-

Write to Node 2:

json
CopyEdit

```
{'key': 'x', 'node_id': 1, 'value': 'B', 'vector_clock': [1, 1, 0]}
```

- 

Read from Node 3:

json
CopyEdit

```
{'key': 'x', 'value': 'B', 'vector_clock': [1, 1, 0]}
```

- 

**Analysis:**

| Operation | Node | Description | Vector Clock |
|---|---|---|---|
| Write A | Node 1 | Local write and broadcast | [1, 0, 0] |
| Write B | Node 2 | Local write after receiving A | [1, 1, 0] |
| Read | Node 3 | Receives A then B in causal order | [1, 1, 0] |

Node 2 applied the write B only after receiving write A, showing causal dependency. Node 3 processed updates in the correct causal order, proving that vector clocks and message buffering are working as intended.

---

## 6. Directory Structure

cpp
CopyEdit

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

---

## 7. Video Demonstration

https://drive.google.com/file/d/1keTH4KqZRD_zhx2mvH0CrnIvFSF3H2R7/view?usp=sharing

---

## 8. Conclusion

The project demonstrates a causally consistent key-value store using vector clocks, ensuring that writes are not applied until their dependencies are satisfied. The architecture and implementation follow distributed systems best practices, with full containerization and verifiable test scenarios.