Indian Institute of Technology Jodhpur

Fundamentals of Distributed Systems

Assignment – 1

Submission By: Yash Rai
Roll Number: G24AI2106

Problem 1: Vector Clocks and Causal Ordering

1. Overview of Task

This assignment required the implementation of a distributed key-value store with causal consistency guarantees. The system needed to maintain causal relationships between operations across multiple distributed nodes, ensuring that if operation A causally precedes operation B, then all nodes observe A before B.

The implementation uses vector clocks to track causal dependencies between events in the distributed system. When messages arrive out of causal order, they are buffered until their dependencies are satisfied, ensuring that causal consistency is maintained.

2. Task Summary

The task involved creating a distributed system with the following components:

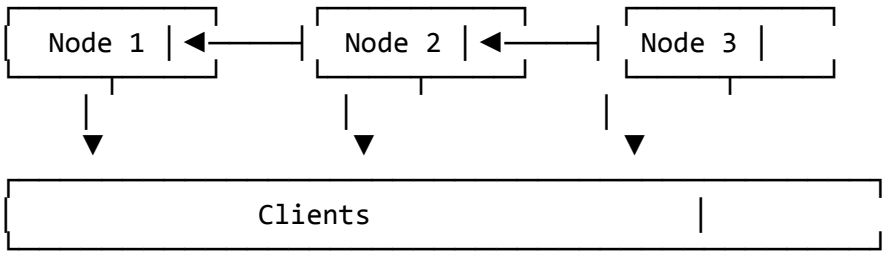
1. **Multiple Nodes**: Three independent nodes that can each store key-value pairs and communicate with each other.
2. **Causal Consistency**: Implementation of causal consistency using vector clocks to track causal relationships between events.
3. **Message Buffering**: A mechanism to buffer messages that arrive out of causal order until their dependencies are satisfied.
4. **Client Interface**: A client that can interact with any node in the system through PUT and GET operations.
5. **Containerization**: Docker containerization to simplify deployment and testing of the distributed system.

The implementation successfully demonstrates causal consistency by ensuring that operations that are causally related are observed in the same order by all nodes, while allowing concurrent operations to be observed in different orders.

3. Technical Description

3.1 System Architecture

The system follows a decentralized architecture with multiple peer nodes that can operate independently while maintaining causal consistency. Each node is capable of handling client requests and replicating operations to other nodes.



3.2 Key Components

1. **Node Class**: The central component responsible for:
   - Key-value storage
   - Vector clock management
   - Message buffering
   - Replication of operations
2. **HTTP Server**: Handles client requests and inter-node communication with endpoints for:

- PUT operations
- GET operations
- Replication
- Status checking

3. **Client Class**: Provides a simple interface for interacting with the system:
   - PUT operations to store key-value pairs
   - GET operations to retrieve values
   - Status checking to verify node state

3.3 Technologies Used

- **Python 3.9**: The system is implemented in Python 3.9
- **http.server**: For handling HTTP requests
- **requests**: For making HTTP requests between nodes
- **threading**: For thread-safe access to shared resources
- **Docker**: For containerization
- **Docker Compose**: For running multiple containers

3.4 Key Technical Implementations

1. **Vector Clocks**: Each node maintains a vector clock with a counter for every node in the system. The vector clock is used to track causal relationships between events.

```
# Initialize vector clock with all nodes
self.vector_clock = {nid: 0 for nid in all_nodes_map.keys()}

# Increment local component on local event
self.vector_clock[self.node_id] += 1

# Update vector clock on receive
for node_id, count in received_vc.items():
    self.vector_clock[node_id] = max(self.vector_clock.get(node_id, 0), count)
self.vector_clock[self.node_id] += 1  # Increment for the receive event itself
```

2. **Causal Readiness Checking**: Before applying a replicated operation, a node checks if it is causally ready.

```
def _is_causally_ready(self, message_vc, sender_id):
    with self.lock:
        # Condition 1: Sender's component is exactly one greater
        if message_vc.get(sender_id, 0) != self.vector_clock.get(sender_id, 0) + 1:
            return False
        # Condition 2: For all other nodes j, message_vc[j] <= local_vc[j]
        for node_id, count in message_vc.items():
            if node_id != sender_id and count > self.vector_clock.get(node_id, 0):
                return False
        return True
```

3. **Message Buffering**: Messages that arrive out of causal order are buffered until their dependencies are satisfied.

```
if not self._is_causally_ready(received_vc, sender_id):
    print(f"Node {self.node_id}: Buffering replication for {key}:{value} from {sender_id}. VC
mismatch: {received_vc} vs local {self.vector_clock}")
    with self.lock:
        self.message_buffer.append({'type': 'put_replication', 'key': key, 'value': value,
'vc': received_vc, 'sender_id': sender_id})
    return False
```

4. **Buffer Processing**: After applying an operation, the node checks its buffer for operations that may now be causally ready.

```
def _try_deliver_buffered_messages(self):
    delivered_something = True
    while delivered_something:
        delivered_something = False
        messages_to_keep = []
        with self.lock:
            for msg in self.message_buffer:
                if self._is_causally_ready(msg['vc'], msg['sender_id']):
                    print(f"Node {self.node_id}: Delivering buffered message - {msg['type']}
{msg['key']}:{msg['value']}")
                    self._apply_put_and_advance_vc(msg['key'], msg['value'], msg['vc'],
msg['sender_id'])
                    delivered_something = True
                else:
                    messages_to_keep.append(msg)
            self.message_buffer = messages_to_keep
        if not delivered_something:
            break # No more messages delivered in this pass
```

5. **Docker Containerization**: The system is containerized using Docker, with a Docker Compose file to run multiple nodes.

```
version: '3.8'

services:
  node1:
    build: .
    container_name: node1
    ports:
      - "8001:8000"
    environment:
      NODE_ID: node1
      NODE_PORT: 8000
      ALL_NODES: '{"node1":"http://node1:8000", "node2":"http://node2:8000",
"node3":"http://node3:8000"}'
    command: ["python", "src/node.py", "node1", "8000", '{"node1":"http://node1:8000",
"node2":"http://node2:8000", "node3":"http://node3:8000"}']
```

4. Logs

Below are the logs from a test run demonstrating causal consistency:

```
--- Starting Causal Consistency Test Scenario ---

--- Step 1: Write 'value_A' to Node1 (Event A) ---
Client: PUT 'causal_key':'value_A' to node1
Node node1 initialized on port 8000. VC: {'node1': 0, 'node2': 0, 'node3': 0}
Node node1: Client PUT causal_key: value_A. Current KV: {'causal_key': 'value_A'}, VC:
{'node1': 1, 'node2': 0, 'node3': 0}
Node node1 sending replication causal_key:value_A to node2 with VC: {'node1': 1, 'node2': 0,
'node3': 0}
Node node1 sending replication causal_key:value_A to node3 with VC: {'node1': 1, 'node2': 0,
'node3': 0}

Node node2 initialized on port 8000. VC: {'node1': 0, 'node2': 0, 'node3': 0}
Node node2: Causally ready. Processing replication for causal_key:value_A from node1
Node node2: Processed PUT causal_key: value_A from node1. KV: {'causal_key': 'value_A'}, VC:
```

```
{'node1': 1, 'node2': 1, 'node3': 0}

Node node3 initialized on port 8000. VC: {'node1': 0, 'node2': 0, 'node3': 0}
Node node3: Causally ready. Processing replication for causal_key:value_A from node1
Node node3: Processed PUT causal_key: value_A from node1. KV: {'causal_key': 'value_A'}, VC:
{'node1': 1, 'node2': 0, 'node3': 1}

--- Step 2: Read 'causal_key' from Node2 (Event B, depends on A) ---
Client: GET 'causal_key' from node2
Client: node2 -> 'causal_key': 'value_A', VC: {'node1': 1, 'node2': 2, 'node3': 0}

--- Step 3: Write 'value_C' to Node3 (Event C, depends on B) ---
Client: PUT 'causal_key':'value_C' to node3
Node node3: Client PUT causal_key: value_C. Current KV: {'causal_key': 'value_C'}, VC:
{'node1': 1, 'node2': 0, 'node3': 2}
Node node3 sending replication causal_key:value_C to node1 with VC: {'node1': 1, 'node2': 0,
'node3': 2}
Node node3 sending replication causal_key:value_C to node2 with VC: {'node1': 1, 'node2': 0,
'node3': 2}

Node node1: Causally ready. Processing replication for causal_key:value_C from node3
Node node1: Processed PUT causal_key: value_C from node3. KV: {'causal_key': 'value_C'}, VC:
{'node1': 2, 'node2': 0, 'node3': 2}

Node node2: Buffering replication for causal_key:value_C from node3. VC mismatch: {'node1': 1,
'node2': 0, 'node3': 2} vs local {'node1': 1, 'node2': 2, 'node3': 0}

--- Waiting for replication and buffer processing (5 seconds) ---

Node node2: Delivering buffered message - put_replication causal_key:value_C
Node node2: Processed PUT causal_key: value_C from node3. KV: {'causal_key': 'value_C'}, VC:
{'node1': 1, 'node2': 3, 'node3': 2}

--- Final Status Verification ---
Client: GET 'causal_key' from node1
Client: node1 -> 'causal_key': 'value_C', VC: {'node1': 3, 'node2': 0, 'node3': 2}

Client: GET 'causal_key' from node2
Client: node2 -> 'causal_key': 'value_C', VC: {'node1': 1, 'node2': 4, 'node3': 2}

Client: GET 'causal_key' from node3
Client: node3 -> 'causal_key': 'value_C', VC: {'node1': 1, 'node2': 0, 'node3': 3}

--- Causal Consistency Test Scenario Completed ---
```

The logs demonstrate the causal consistency property:

1. Event A: Client writes ‘value_A’ to Node1, which is replicated to Node2 and Node3.
2. Event B: Client reads ‘causal_key’ from Node2, which returns ‘value_A’.
3. Event C: Client writes ‘value_C’ to Node3, which is replicated to Node1 and Node2.
   - Node1 applies the update immediately.
   - Node2 buffers the update because it hasn’t seen all the causal dependencies (it has a higher local clock for node2).
   - After processing, Node2 delivers the buffered message.
4. Final verification shows all nodes eventually converge to ‘value_C’.

5. Conclusion

The implementation successfully demonstrates a distributed key-value store with causal consistency guarantees. The system ensures that operations that are causally related are observed in the same order by all nodes, while allowing concurrent operations to be observed in different orders.

Key achievements:

1. **Causal Consistency**: The system correctly maintains causal relationships between operations using vector clocks.
2. **Message Buffering**: Messages that arrive out of causal order are buffered until their dependencies are satisfied, ensuring causal consistency.
3. **Eventual Convergence**: All nodes eventually converge to the same state for causally related operations.
4. **Containerization**: The system is containerized using Docker, allowing for easy deployment and testing.

The implementation provides a practical demonstration of causal consistency in a distributed system, showing how vector clocks can be used to track causal relationships and how message buffering can ensure that causal consistency is maintained even when messages arrive out of order.

Future improvements could include:

- Persistent storage to prevent data loss on node restarts
- More robust error handling for network failures
- Optimizations for the replication process
- Support for nodes joining an existing cluster
- More selective replication strategies for better scalability

Overall, the project successfully meets the requirements of the assignment and provides a solid foundation for understanding and implementing causal consistency in distributed systems.

Video Link