**D17B1/Batch B/Sanjana Asrani**

**Aim**: To understand the concepts of distributed consistency management in distributed systems and to implement and observe different consistency models.

**Theory**:  
Introduction to Distributed Consistency Management:

In distributed systems, consistency management refers to the techniques and mechanisms used to ensure that all copies of data stored across multiple nodes in the system remain consistent. Achieving consistency in a distributed environment is challenging due to factors such as network delays, node failures, and concurrent updates. Consistency management is essential for maintaining data integrity, ensuring correct operation of distributed applications, and providing predictable behavior to users.

**Key Challenges in Distributed Consistency Management:**

1. Network Delays: Communication delays between nodes can lead to inconsistencies in data propagation and synchronization.

2. Concurrency Control: Concurrent updates to the same data from multiple nodes can result in conflicts and inconsistencies if not properly managed.

3. Node Failures: Failures of nodes in the distributed system can disrupt data replication and synchronization processes, leading to inconsistencies.

4. Scalability: Consistency management techniques must scale efficiently with the size of the distributed system to handle increasing loads and data volumes.

Consistency Models in Distributed Systems:

1. Strong Consistency: In a strongly consistent system, all read operations return the most recent write value, ensuring that all nodes see the same consistent view of the data at all times. Achieving strong consistency often requires coordination and synchronization mechanisms such as locking and two-phase commit protocols.

2. Eventual Consistency: Eventual consistency allows for temporary inconsistencies between replicas but guarantees that eventually, all replicas will converge to the same value. This model is often used in distributed systems where low-latency communication and high availability are prioritized over immediate consistency.

3. Causal Consistency: Causal consistency preserves causal relationships between related operations, ensuring that causally dependent operations are ordered correctly across replicas. This model provides a balance between strong consistency and eventual consistency by relaxing consistency requirements based on causality.

4. FIFO Consistency: FIFO (First-In-First-Out) consistency ensures that the order of write operations is preserved across replicas. Writes are applied in the order they were received by each replica, maintaining consistency based on the order of operations.

Data-Centric Consistency Models:

Data-centric consistency models focus on ensuring consistency at the level of data objects or records. These models prioritize consistency guarantees for individual data items and provide mechanisms to enforce consistency constraints across distributed replicas.

1. Sequential Consistency: Sequential consistency requires that all operations on a data object appear to be executed in the same order for all processes. This model ensures that the result of any execution is the same as if the operations by all processes were executed in some sequential order, preserving consistency based on the global order of operations.

2. Linearizability: Linearizability extends sequential consistency by ensuring that each operation appears to take effect instantaneously at some point between its invocation and response. This model provides stronger consistency guarantees by imposing real-time ordering constraints on operations, ensuring that the system behaves as if there is a single global timeline for all operations.

3. FIFO Consistency: FIFO consistency, also known as causal consistency, ensures that the order of write operations is preserved across replicas. Writes are applied in the order they were received by each replica, maintaining consistency based on the causal relationship between operations.

Client-Centric Consistency Models:

Client-centric consistency models focus on providing consistency guarantees based on the behavior and expectations of individual clients or users interacting with the distributed system. These models prioritize consistency at the client interface level, ensuring that clients perceive consistent behavior and state transitions.

1. Monotonic Read/Write Consistency: Monotonic consistency ensures that if a client reads the value of a data object, it will always see at least that value or a more recent one in subsequent reads. Similarly, monotonic write consistency ensures that if a client writes a value to a data object, the value will eventually be visible to all subsequent reads by the same client.

2. Read-Your-Writes Consistency: Read-your-writes consistency ensures that if a client performs a write operation on a data object, all subsequent read operations by the same client on the same or related data objects will reflect the effects of the write. This model provides stronger consistency guarantees by preserving the causal relationship between client write and read operations.

**Practical**:

1. *Data Centric FIFO consistency (dc08a)*

Code:

class FIFOConsistencyDatabase:

    def \_\_init\_\_(self):

        self.data = {}

        self.write\_queue = {}

    def read(self, key):

        values = self.\_read\_with\_fifo\_consistency(key)

        latest\_values = {}

        for value, writer\_id in values:

            latest\_values[key] = (value, writer\_id)

        return list(latest\_values.values())

    def write(self, key, value, process\_id):

        for db in databases:

            db.data[key] = (value, process\_id)

            db.write\_queue.setdefault(process\_id, []).append(key)

    def \_read\_with\_fifo\_consistency(self, key):

        values = []

        for process\_id in self.write\_queue:

            for k in self.write\_queue[process\_id]:

                if k == key:

                    value, writer\_id = self.data[k]

                    values.append((value, writer\_id))

        values.sort(key=lambda x: self.write\_queue[x[1]].index(key))

        return values

def write\_values(db, values, process\_id):

    global values\_order

    values\_order = values

    for value in values:

        db.write(value, f'{value} by process{process\_id}', process\_id)

    print(f"\nProcess{process\_id} writes {' and '.join(values)}")

def read\_values(db, process\_id):

    print(f"\nProcess{process\_id} reads:")

    for value in values\_order:

        values = db.read(value)

        if values:

            for data, writer\_id in values:

                print(f" Value: {data} by process{writer\_id} written by Process{writer\_id}")

        else:

            print(f" No current values for the given key")

def simulate\_processes(databases):

    num\_processes = len(databases)

    write\_values(databases[0], ['valueA', 'valueB'], 1)

    read\_values(databases[1], 2)

    read\_values(databases[2], 3)

    write\_values(databases[2], ['valueA', 'valueB'], 3)

    read\_values(databases[1], 2)

    write\_values(databases[3], ['valueB', 'valueA'], 4)

    read\_values(databases[0], 1)

    read\_values(databases[1], 2)

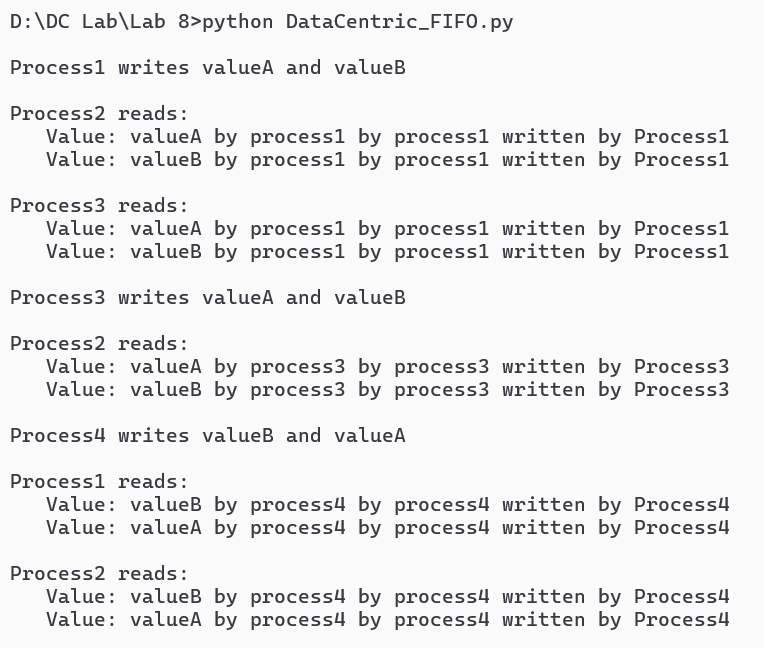
# Create databases for each process

num\_processes = 4

databases = [FIFOConsistencyDatabase() for \_ in range(num\_processes)]

# Simulate processes

simulate\_processes(databases)

Output:  


1. *Client Centric Read-Your-Write consistency (dc08b)*

Code:

class ClientCentricReadWriteConsistencyDatabase:

    def \_\_init\_\_(self):

        self.data = {}

        self.client\_history = {}

    def read(self, key):

        if key in self.data:

            return self.data[key]

        else:

            return None

    def write(self, key, value, client\_id):

        if key not in self.client\_history:

            self.client\_history[key] = []

        self.client\_history[key].append((client\_id, value))

        self.data[key] = (client\_id, value)

def main():

    db\_rw = ClientCentricReadWriteConsistencyDatabase()

    while True:

        print("\nOptions:")

        print("1. Write data")

        print("2. Read data")

        print("3. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':

            key = input("Enter key: ")

            value = input("Enter value: ")

            client\_id = input("Enter client ID: ")

            db\_rw.write(key, value, client\_id)

            print("Data written successfully!")

        elif choice == "2":

            key = input("Enter key to read: ")

            value = db\_rw.read(key)

            if value is not None:

                client\_id, val = value

                print(f"value for key '{key}' written by client {client\_id}: {val}")

            else:

                print(f"No value found for key '{key}'")

        elif choice == "3":

            print("Exiting...")

            break

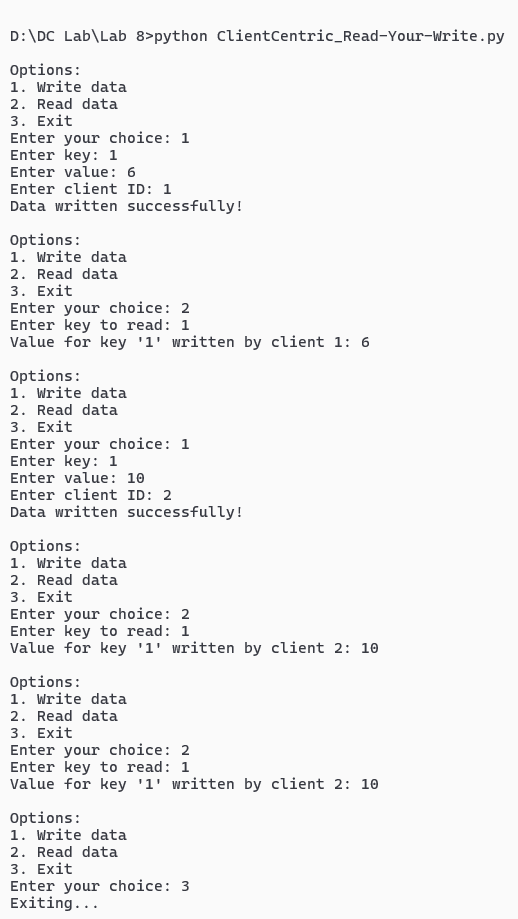
        else:

            print("Invalid choice. Please try again.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

Output:



**Conclusion**:

Distributed consistency management encompasses a range of models and techniques aimed at ensuring data consistency across distributed nodes in a system. By understanding the challenges and trade-offs involved in achieving consistency, developers can design effective consistency models and implement mechanisms to ensure data integrity, reliability, and predictability in distributed applications. The choice of consistency model depends on factors such as application requirements, performance considerations, and the desired balance between consistency guarantees and system scalability.