PARALLEL AND DISTRIBUTION COMPUTING

Y. Charishma
221FA04252
CSE SECTION -B, BATCH-11
VIGNAN UNIVERSITY
Tenali, India
cherry1780677@gmail.com

B. Bhanu Latha
221FA04448
CSE SECTION -B, BATCH-11
VIGNAN UNIVERSITY
Tenali, India
bhanulathabalisetti@gmail.com

P. Sai Vedagna
221FA04627
CSE SECTION-B, BATCH-11
Vignan University
Bapatla,India
vedagna1424@gmail.com

B.Sruthi
221FA04666
CSE SECTION-B, BATCH-11
Vignan University
Jangaon, India
sruthibajjuri04@gmail.com

Abstract— **High availability and fault tolerance are critical in distributed systems. This paper explores the implementation of data replication techniques, such as primary-backup and quorum-based replication, to ensure system** reliability **against node failures. Additionally, a checkpointing mechanism is integrated to periodically save system states, allowing seamless recovery after failures. Finally, we analyze the overhead introduced by replication, checkpointing, and recovery mechanisms and their impact on overall system latency and throughput.**

I. QUESTION

A. Design a system that uses data replication techniques such as primary-backup or quorum-based replication to ensure high availability despite node failures.
B. Implement a check pointing mechanism where system states are periodically saved so that computations can resume from the last checkpoint after a failure.
C. Analyze the overhead introduced by replication, check pointing, and recovery mechanisms and their effect on overall system latency and throughput.

II. INTRODUCTION

Distributed systems are essential for handling large-scale applications, but they are inherently vulnerable to failures caused by hardware malfunctions, network disruptions, or software crashes. These failures can lead to data loss, downtime, and reduced system efficiency. To enhance fault tolerance and maintain high availability, systems employ replication and checkpointing techniques. Replication ensures that multiple copies of data are distributed across different nodes, preventing single points of failure. Depending on the strategy used, such as primary-backup or quorum-based replication, systems can achieve different levels of consistency, fault tolerance, and performance optimization.

Checkpointing complements replication by periodically saving the system state, enabling applications to recover from the last saved state instead of restarting from scratch. This technique minimizes data loss and reduces downtime after a failure, making it particularly valuable for long-running computations and real-time processing. However, both replication and checkpointing introduce additional computational and storage overhead. This study explores the integration of these techniques in a distributed system, analyzing their trade-offs in terms of performance, resource consumption, and overall system reliability.

III. WORKFLOW

A. System Design for High Availability Using Replication

In a distributed system, data replication is essential to ensure high availability and fault tolerance in case of node failures. There are two common replication techniques:

1. Primary-Backup Replication

This is a widely used method where one node (the primary) handles all read and write requests, while one or more backup nodes maintain copies of the data.

How It Works:

- The primary node processes client requests and updates the backup nodes.

- Backups synchronously (immediate update) or asynchronously (delayed update) replicate the primary's state.
- If the primary node fails, a backup node takes over and becomes the new primary.

Advantages:
✅ Fast recovery after failure.
✅ Strong consistency since all updates originate from the primary.
✅ Simple and easy to implement.

Disadvantages:
⊘ Increased write latency due to synchronization between primary and backups.
⊘ The system might slow down if the primary **is** overloaded.

2. Quorum-Based Replication

Instead of relying on a single primary, this approach distributes requests among multiple nodes using a quorum-based voting system.

How It Works:

- A quorum is a subset of nodes that must agree on a request before it is processed.
- Two quorum values are defined:
  - **Read quorum (R):** The minimum number of nodes needed to confirm a read request.
  - **Write quorum (W):** The minimum number of nodes needed to acknowledge a write request.
- The system follows the rule **R + W > Total Nodes** to ensure that at least one node always has the most recent update.

Example:
For a 5-node system, you could set:

- **R = 2** (at least 2 nodes must confirm a read).
- **W = 3** (at least 3 nodes must confirm a write).

Advantages:
✅ More scalable than primary-backup, as it allows multiple node**s** to handle requests.
✅ No single point of failure.
✅ Works well for large-scale distributed databases.

Disadvantages:
⊘ Higher read/write latency as multiple nodes must communicate.
⊘ Managing consistency can be complex.

✅ **Best Use Cases:**

- **Primary-backup** is best for small-scale critical systems (e.g., banking transactions).
- **Quorum-based replication** is ideal for distributed, global-scale applications (e.g., cloud databases, blockchain).

B. Checkpointing Mechanism for System Recovery

Checkpointing is a technique used to periodically save system state so that computations can resume from the last checkpoint after a failure.

1. Types of Checkpointing:

Periodic Checkpointing:

- The system saves its state at fixed intervals (e.g., every 10 minutes).
- Simple but may waste resources if failure happens just before the next checkpoint.

Incremental Checkpointing:

- Instead of saving the full state each time, it saves only the changes since the last checkpoint.
- Reduces storage and computation costs.

Application-Aware Checkpointing:

- The application determines when and what to save, ensuring efficient resource usage.
- Example: A database saves only uncommitted **transactions** rather than the entire dataset.

2. Checkpoint Recovery Process:

- When a failure occurs, the system retrieves the last saved checkpoint.
- It reapplies pending operations (if any) to restore consistency.
- The system resumes normal operation from that point, minimizing downtime and data loss.

Advantages:
✅ Reduces downtime by allowing fast recovery.
✅ Prevents full recomputation of lost processes.

Disadvantages:
⊘ Requires additional storage and processing overhead.
⊘ More frequent checkpointing increases system load.

✅ **Best Use Cases:**

- Supercomputing systems (e.g., weather simulations).
- Databases and cloud computing (e.g., AWS EC2 snapshots).

C. Analysis of Overhead Introduced by Replication, Checkpointing, and Recovery Mechanisms

- **Replication Overhead:**
  - Primary-Backup: High write latency (due to synchronization) but fast reads.
  - Quorum-Based: Increased network and coordination overhead but better fault tolerance.
  - Effect: Higher write latency, potential throughput reduction.
- **Checkpointing Overhead:**
  - Periodic Checkpointing: High I/O and storage overhead, affecting system response time.
  - Incremental Checkpointing: Lower overhead but complex state management.
  - Effect: Temporary latency spikes and reduced throughput during checkpointing.
- **Recovery Overhead:**
  - Checkpoint-Based Recovery: Restoring state takes time, impacting system availability.
  - Primary Failover: Delay in leader election affects request processing.
  - Effect: Increased downtime, reduced throughput post-recovery.

IV. IMPLEMENTATION

1) Primary-Backup Replication

```
import threading
import time

class PrimaryBackup:
    def __init__(self):
        self.primary = "Node A"
        self.backup = "Node B"
        self.data = {}

    def write_data(self, key, value):
        self.data[key] = value
        print(f"Primary ({self.primary}): Updated {key} = {value}")
        self.replicate_to_backup()

    def replicate_to_backup(self):
        print(f"Backup ({self.backup}): Synchronized data: {self.data}")

    def failover(self):
        self.primary, self.backup = self.backup, self.primary
        print(f"Failover occurred. New Primary: {self.primary}")

system = PrimaryBackup()
system.write_data("x", 10)
time.sleep(2)
system.failover()
system.write_data("y", 20)
```

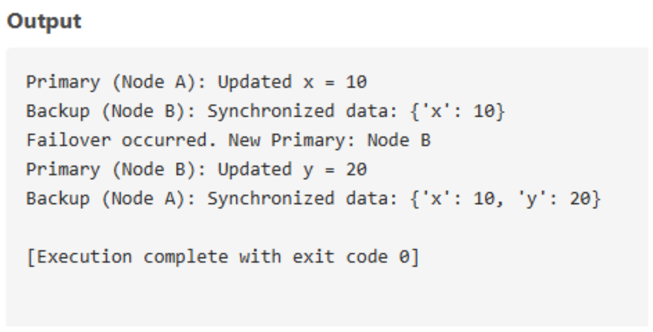
2) Checkpointing mechanism

```
import pickle
import time

class CheckpointSystem:
    def __init__(self):
        self.state = {}  # Dictionary to store system state

    def update_state(self, key, value):
        """Update the system state with new values."""
        self.state[key] = value
        print(f"State updated: {key} = {value}")

    def save_checkpoint(self, filename="checkpoint.pkl"):
        """Save the current state to a checkpoint file."""
        with open(filename, "wb") as f:
            pickle.dump(self.state, f)
        print("Checkpoint saved.")

    def load_checkpoint(self, filename="checkpoint.pkl"):
        """Load the last saved state from the checkpoint file."""
        try:
            with open(filename, "rb") as f:
                self.state = pickle.load(f)
            print("Checkpoint loaded:", self.state)
        except FileNotFoundError:
            print("No checkpoint found. Starting fresh.")

# Simulating system execution
system = CheckpointSystem()
system.load_checkpoint()  # Load previous state (if any)

# Updating state and saving checkpoints at intervals
system.update_state("progress", 50)
system.save_checkpoint()
time.sleep(2)

system.update_state("progress", 75)
system.save_checkpoint()
```

```
time.sleep(2)

# Simulating a system failure and recovery
print("\nSimulating system failure...\nRestarting system...\n")
recovered_system = CheckpointSystem()
recovered_system.load_checkpoint()  # Recover from the last checkpoint
```

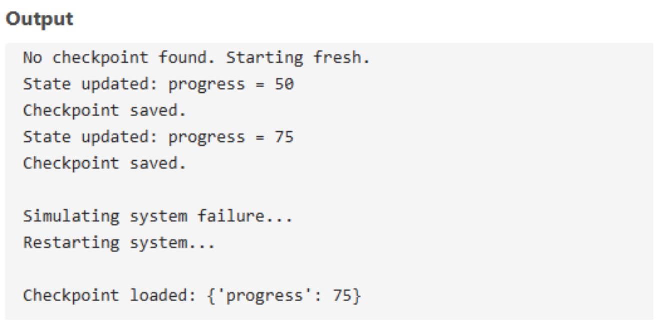
V. INPUT AND OUTPUTS

1.


Output

```
Primary (Node A): Updated x = 10
Backup (Node B): Synchronized data: {'x': 10}
Failover occurred. New Primary: Node B
Primary (Node B): Updated y = 20
Backup (Node A): Synchronized data: {'x': 10, 'y': 20}

[Execution complete with exit code 0]
```

2.)


Output

```
No checkpoint found. Starting fresh.
State updated: progress = 50
Checkpoint saved.
State updated: progress = 75
Checkpoint saved.

Simulating system failure...
Restarting system...

Checkpoint loaded: {'progress': 75}
```

VI. CONCLUSION

Ensuring high availability and fault tolerance in distributed systems requires a strategic combination of replication, checkpointing, and recovery mechanisms. Replication techniques like primary-backup and quorum-based replication help maintain multiple copies of data across nodes, reducing the risk of data loss during failures. However, these techniques introduce additional network and storage overhead, impacting write latency and overall system throughput. Checkpointing further enhances fault tolerance by periodically saving system states, allowing computations to resume from the last saved state instead of restarting. While this minimizes downtime and prevents significant data loss, it increases I/O and computation overhead, affecting system performance. Recovery mechanisms, such as failover handling and state restoration, ensure that services continue running even after a failure. However, failover processes can introduce temporary latency spikes, and restoring a system from a checkpoint may slow down initial execution. The trade-offs between high availability and system efficiency depend on factors like checkpoint frequency, replication consistency, and failover speed. A well-designed system must balance performance, resource utilization, and fault tolerance to minimize downtime while maintaining efficiency. By optimizing these techniques, distributed systems can provide reliable, scalable, and resilient services despite node failures.

****