UNIT4

Fault tolerance Techniques in Computer System

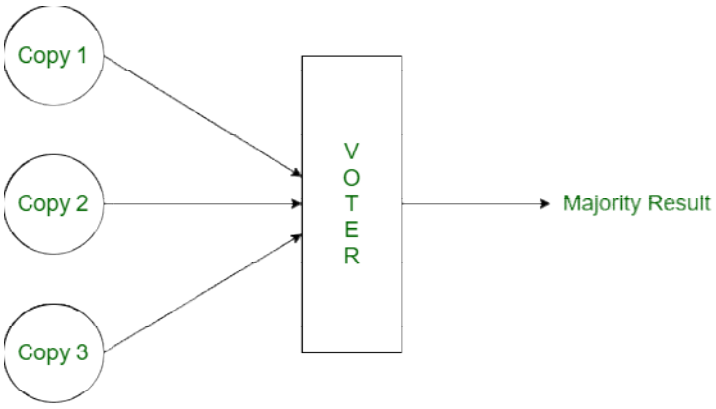
Fault tolerance is the process of working of a system in a proper way in spite of the occurrence of the failures in the system. Even after performing the so many testing processes there is possibility of failure in system. Practically a system can't be made entirely error free. hence, systems are designed in such a way that in case of error availability and failure, system does the work properly and given correct result.
Any system has two major components – Hardware and Software. Fault may occur in either of it. So there are separate techniques for fault tolerance in both hardware and software.

Hardware Fault tolerance Techniques:

Making a hardware fault tolerance is simple as compared to software. Fault tolerance techniques make the hardware work proper and give correct result even some fault occurs in the hardware part of the system. There are basically two techniques used for hardware fault tolerance:

1. **BIST** –
   BIST stands for Build in Self Test. System carries out the test of itself after a certain period of time again and again, that is BIST technique for hardware fault tolerance. When system detects a fault, it switches out the faulty component and switches in the redundant of it. System basically reconfigure itself in case of fault occurrence.

2. **TMR** –
   TMR is Triple Modular Redundancy. Three redundant copies of critical components are generated and all these three copies are run concurrently. Voting of result of all redundant copies are done and majority result is selected. It can tolerate the occurrence of a single fault at a time.

3. 

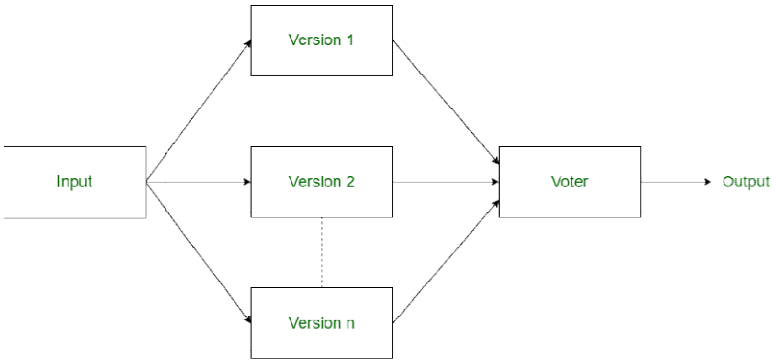
   TMR

Software Fault tolerance Techniques:

Software fault tolerance techniques are used to make the software reliable in the condition of fault occurrence and failure. There are three techniques used in software fault tolerance. First two techniques are common and are basically an adaptation of hardware fault tolerance techniques.

1. **N version Programming –**

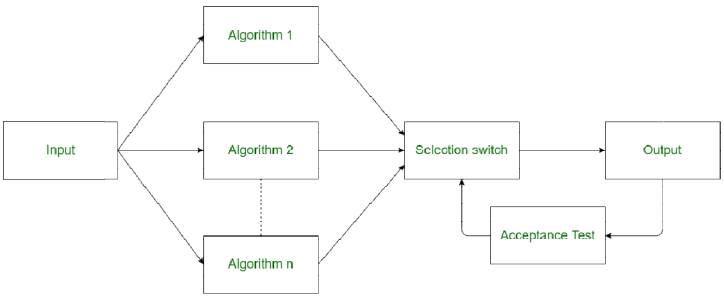
In N version programming, N versions of software are developed by N individuals or groups of developers. N version programming is just like TMR in hardware fault tolerance technique. In N version programming, all the redundant copies are run concurrently and result obtained is different from each processing. The idea of N version programming is basically to get the all errors during development only.



N-version Programming

2. **Recovery Blocks –**

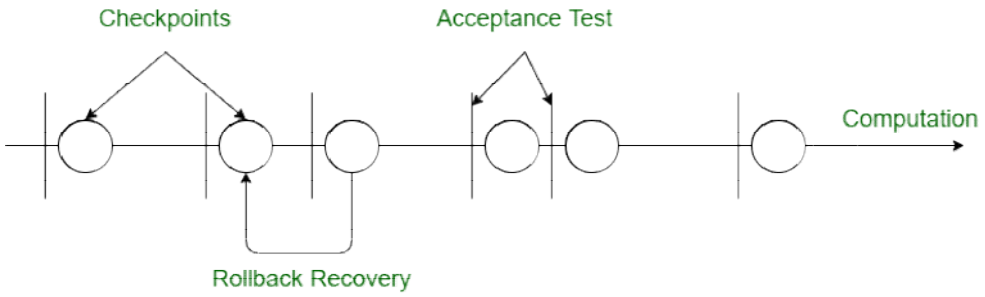
Recovery blocks technique is also like the N version programming but in recovery blocks technique, redundant copies are generated using different algorithms only. In recovery block, all the redundant copies are not run concurrently and these copies are run one by one. Recovery block technique can only be used where the task deadlines are more than task computation time.



Recovery Blocks

3. **Checkpointing and Rollback Recovery –**

This technique is different from above two techniques of software faulttolerance. In this technique, system is tested each time when we perform some computation. This techniques is basically useful when there is processor failure or data corruption.



Failure recovery

Failure recovery and fault tolerance are both critical concepts in system design, particularly for ensuring the reliability, availability, and robustness of systems. Although related, they refer to slightly different approaches to dealing with system failures.

1. Failure Recovery

Failure recovery focuses on responding to system failures after they occur, aiming to restore normal operations. The key steps involve detecting the failure, diagnosing the problem, and taking corrective action to resume normal operation. Failure recovery often involves:

- **Backup and Restore:** Systems periodically back up data, allowing a recovery from the last stable state after a failure (e.g., database backups, cloud backup services).
- **Checkpointing:** In some systems, checkpoints are periodically saved, so when a failure occurs, the system can restart from a known good state rather than from scratch.
- **Rollback and Rollforward:** In some cases, systems can roll back to a previous state (undoing operations) or roll forward by replaying transactions from logs to bring the system back to a consistent state.
- **Automatic Restart:** Some systems have mechanisms to automatically restart processes or services that have failed, minimizing downtime.

Examples of Failure Recovery:

- Database Systems: Many databases implement logbased recovery mechanisms that allow restoring data to a consistent state after a crash.

- Operating Systems: Some OSes have mechanisms like file system journaling, which allows them to recover from unexpected shutdowns without data corruption.

2. Fault Tolerance

Fault tolerance, on the other hand, refers to the system's ability to continue operating correctly even when one or more components fail. Instead of waiting for a failure to occur and then recovering, faulttolerant systems are designed to prevent failures from causing service disruptions by providing redundancy and ensuring the system can handle failures in real time.

Key concepts in fault tolerance include:

- Redundancy: Systems maintain extra copies of critical components (hardware, software, or data) so that if one component fails, another can immediately take over. This can be achieved through techniques like:
- Replication: Replicating data or processes across multiple nodes or systems (e.g., distributed databases, RAID storage).
- Failover Mechanisms: Automatic switching to a backup system, component, or node in case of failure.
- Error Detection and Correction: Mechanisms like checksums or parity bits are used to detect and sometimes correct errors in data transmission or storage.
- Graceful Degradation: Some systems are designed to reduce functionality in case of failure, but still provide core services. This is common in realtime systems where it’s better to continue running with reduced capacity than to fail completely.

Examples of Fault Tolerance:

- Distributed Systems: Cloud services like AWS or Google Cloud use multiple data centers and nodes, so if one node fails, another can handle the workload without service disruption.
- Aviation Systems: Airplanes often have redundant hardware systems (like multiple engines or backup controls) to ensure they can still operate safely in case of a failure.

Key Differences:

Timing of Response:

- Failure Recovery: Response happens after a failure.
- Fault Tolerance: Systems are designed to prevent service disruption during a failure.

Goal:

- Failure Recovery: To restore normal system operation postfailure.
- Fault Tolerance: To continue normal operation without noticeable impact.

Recovery With Concurrent Transactions

Concurrency control means that multiple transactions can be executed at the same time and then the interleaved logs occur. But there may be changes in transaction results so maintain the order of execution of those transactions.

During recovery, it would be very difficult for the recovery system to backtrack all the logs and then start recovering.

Recovery with concurrent transactions can be done in the following four ways.

1. Interaction with concurrency control
2. Transaction rollback
3. Checkpoints
4. Restart recovery

Interaction with concurrency control :
In this scheme, the recovery scheme depends greatly on the concurrency control scheme that is used. So, to rollback a failed transaction, we must undo the updates performed by the transaction.

Transaction rollback :

- In this scheme, we rollback a failed transaction by using the log.
- The system scans the log backward a failed transaction, for every log record found in the log the system restores the data item.

Checkpoints :

- Checkpoints is a process of saving a snapshot of the applications state so that it can restart from that point in case of failure.
- Checkpoint is a point of time at which a record is written onto the database form the buffers.
- Checkpoint shortens the recovery process.
- When it reaches the checkpoint, then the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction till the next checkpoint and so on.
- The checkpoint is used to declare the point before which the DBMS was in the consistent state, and all the transactions were committed.

To ease this situation, 'Checkpoints' Concept is used by the most DBMS.

- In this scheme, we used checkpoints to reduce the number of log records that the system must scan when it recovers from a crash.
- In a concurrent transaction processing system, we require that the checkpoint log record be of the form <checkpoint L>, where **'L'** is a list of transactions active at the time of the checkpoint.
- A fuzzy checkpoint is a checkpoint where transactions are allowed to perform updates even while buffer blocks are being written out.

Restart recovery :

- When the system recovers from a crash, it constructs two lists.
- The undolist consists of transactions to be undone, and the redolist consists of transaction to be redone.
- The system constructs the two lists as follows: Initially, they are both empty. The system scans the log backward, examining each record, until it finds the first <checkpoint> record.

Consistent set of checkpoints

A consistent set of checkpoints refers to a coordinated series of saved states across various components of a distributed or multiprocess system, where each checkpoint reflects a state that is consistent across all components at a particular point in time. This concept is crucial in distributed systems and parallel computing to ensure that, in the event of a failure, the system can be restored to a valid state without inconsistencies.

Why is Consistency Important?

In distributed or multithreaded systems, different components or processes operate simultaneously, potentially on different physical machines. If checkpoints are saved independently by each process without coordination, one process might save a state before a critical event (like receiving a message), while another saves a state after the event. This inconsistency can lead to problems like lost messages, unfinished transactions, or even deadlocks when the system tries to recover from a failure.

A consistent checkpoint ensures that all parts of the system are in a logically consistent state—no events or messages are partially processed or missing—and recovery can happen without introducing inconsistencies.

Characteristics of a Consistent Set of Checkpoints

- Global Consistency: The state of each individual component or process aligns with the states of others, ensuring the system can resume operation correctly from that point. In distributed systems, this means that the states of different nodes reflect the same "moment" in logical time.
- No Orphan Messages: When recovering from a checkpoint, the system must not have messages that were sent after the checkpoint but received before it.
- No Lost Updates: If a checkpoint is restored, no updates made to the system before the checkpoint should be lost or uncommitted.

How to Create a Consistent Set of Checkpoints

There are several strategies used in distributed systems to ensure that a set of checkpoints is consistent:

1. Coordinated Checkpointing

In coordinated checkpointing, processes or nodes in the system coordinate to ensure they all create checkpoints at the same logical point in time. The idea is to avoid inconsistencies between the states of different processes by synchronizing the checkpointing operation.

Global Synchronization: All processes are paused and checkpoints are saved at the same time. This ensures that no communication is happening during the checkpoint, creating a globally consistent snapshot.

TwoPhase or ThreePhase Protocols: These protocols involve message exchanges between processes to ensure that all participants agree on a common checkpoint time. The protocol ensures that if any process fails to complete its checkpoint, no checkpoints are committed.

Pros:

Guarantees consistency.

Simple to implement in terms of recovery.

Cons:

Can introduce performance overhead due to synchronization.

Can lead to systemwide delays, especially in largescale distributed systems.

2. Uncoordinated Checkpointing

In uncoordinated checkpointing, each process independently saves checkpoints at its own pace, without synchronizing with other processes. However, this can lead to inconsistent states where some processes have checkpoints from before a critical event, while others have checkpoints from after.

This method typically relies on additional mechanisms during recovery, such as message logging to replay events and ensure consistency after the fact. It might involve rollback propagation, where some processes need to roll back further than others to ensure a consistent recovery.

Pros:

No synchronization overhead, allowing higher performance in normal operation.

Cons:

Recovery becomes complex and may require rolling back multiple processes to earlier checkpoints to achieve consistency (this is called the domino effect).

3. ChandyLamport Algorithm (for Distributed Systems)

This is a wellknown algorithm used to create a consistent global state (and thus a consistent set of checkpoints) in distributed systems. It works by propagating a "marker" message throughout the system, and each process takes a local checkpoint when it receives the marker for the first

time. The algorithm ensures that all processes' checkpoints form a consistent global snapshot without requiring them to stop all activities simultaneously.

How it works:

1. One process initiates the checkpointing by sending a marker message to all other processes it communicates with.

2. When a process receives the marker, it saves its local state and starts recording messages received from other processes (before their marker is received).

3. After the marker is received from all other processes, the process saves its recorded messages as part of the checkpoint.

This algorithm ensures that the system has a consistent snapshot without requiring synchronization.

4. Incremental Checkpointing

Instead of saving the entire state of the system at each checkpoint, incremental checkpointing saves only the changes made since the last checkpoint. This can reduce the performance impact of checkpointing, especially in systems where the state is large, but changes between checkpoints are small.

Recovery Using Consistent Checkpoints

When a failure occurs, the system can roll back to the last consistent set of checkpoints across all processes or nodes. This ensures that the system resumes from a consistent, welldefined state, and no data is lost or inconsistencies are introduced.

Advantages of Consistent Checkpoints:

- Guaranteed System State Integrity: Ensures that the system resumes without inconsistencies, making recovery reliable.
- Reduced Complexity in Recovery: Recovery is simpler and faster because there is no need to replay or recompute events.

Challenges:

- Performance Overhead: Coordinating checkpoints or ensuring consistency can slow down system performance, especially in highfrequency checkpointing.
- Scalability: As systems grow larger, coordinating checkpoints or maintaining consistency across many nodes or processes can become more difficult.

Synchronous and Asynchronous checkpointing and recovery

Synchronous and asynchronous checkpointing refer to different approaches to saving the state of a system (or its components) in the context of fault tolerance and recovery. These techniques are used to ensure that, in the event of a failure, the system can be restored to a previous consistent state. Here's a breakdown of both approaches, their advantages, disadvantages, and how they affect system recovery:

1. Synchronous Checkpointing

In synchronous checkpointing, processes or components in a system coordinate to save their states simultaneously. This ensures that the checkpoint represents a consistent global state across the system, making recovery easier and more straightforward.

How Synchronous Checkpointing Works:

- Global Coordination: Before taking a checkpoint, all participating processes or nodes in the system coordinate to pause their activities. A checkpoint is taken only after ensuring that no process is in the middle of sending or receiving messages that could lead to inconsistency.
- Barrier Synchronization: Some systems use a barrier, a synchronization point where all processes wait until every process has reached that point before they continue execution.
- Snapshot: The states of all processes are saved at this "synchronized" point, creating a globally consistent snapshot of the system.

Advantages:

Guaranteed Consistency: Because the system pauses and checkpoints simultaneously, there is no need to worry about missing messages or partial updates. The entire system is captured in a consistent state.

Simpler Recovery: During recovery, the system can simply restore the checkpointed state without needing additional logs or rollbacks. There are no complications like message reordering or lost events.

Disadvantages:

Performance Overhead: Synchronous checkpointing introduces delays because all processes must stop and wait for each other. This can be problematic in systems with many nodes, where synchronization can lead to significant performance penalties.

Scalability Issues: As the number of processes grows, the time to coordinate a global checkpoint increases, making this method less scalable for large distributed systems.

Recovery in Synchronous Checkpointing:

Since all processes have synchronized checkpoints, recovery is straightforward. After a failure, the system is rolled back to the most recent global checkpoint. No need for further coordination or replaying messages because the checkpoint is already consistent across all components.

2. Asynchronous Checkpointing

In asynchronous checkpointing, each process or node independently saves its state without coordinating with other processes. This allows checkpoints to be taken at different times without halting the entire system, improving performance and scalability.

How Asynchronous Checkpointing Works:

- Independent Checkpoints: Each process or node takes checkpoints independently of others, based on its own timing or predefined intervals.
- Message Logging (Optional): To handle potential inconsistencies between checkpoints, message logging is often used to record the messages exchanged between processes. This ensures that, during recovery, lost or unprocessed messages can be replayed to bring the system back to a consistent state.
- No Barrier Synchronization: Processes do not wait for each other to take a checkpoint. Each one continues operating without pausing or coordinating with the others.

Advantages:

Reduced Performance Overhead: Since processes don't need to pause and coordinate with each other, the system can continue operating with minimal disruption. This is especially useful in large distributed systems.

Better Scalability: Asynchronous checkpointing scales better in large systems because there's no global coordination. Each process can checkpoint independently, without waiting for others.

Disadvantages:

Inconsistent Checkpoints: Because processes save their states independently, checkpoints might not reflect a consistent global state. One process might checkpoint before receiving a message, while another might checkpoint after sending it, leading to an inconsistent system state.

Complex Recovery: Recovery from asynchronous checkpoints is more complicated. The system may need to roll back some processes to earlier checkpoints and replay messages to ensure consistency, which can lead to the domino effect (where multiple rollbacks are required).

Recovery in Asynchronous Checkpointing:

- As checkpoints are not globally consistent, recovery typically requires additional mechanisms like message logging or rollback propagation.
- Message Logging: Messages between processes are logged, and during recovery, the system can replay the logged messages to bring all processes back to a consistent state.
- Rollback Propagation: If an inconsistency is detected during recovery (e.g., one process has a checkpoint from before a message was received, and another from after it was sent), some processes might need to roll back to earlier checkpoints. This can sometimes lead to a cascading effect, known as the domino effect, where multiple processes need to rollback until a consistent state is found.

Voting protocol in distributed system

In distributed systems, a voting protocol is a mechanism used to ensure that a group of distributed processes or nodes can agree on a single decision or state, even in the presence of faults or network issues. Voting protocols are essential for achieving consensus, coordinating actions, and ensuring data consistency across a distributed network.

There are several types of voting protocols in distributed systems, with different approaches based on the system's needs and tolerance for failures. These protocols are commonly used in databases, distributed transaction systems, and faulttolerant services like replication, leader election, and distributed consensus.

Key Elements of Voting Protocols

- **Consensus:** The protocol ensures that all participants (nodes, processes, or replicas) in the distributed system agree on a common decision or outcome.
- **Quorum:** A quorum is the minimum number of participants that must agree to reach a decision. Quorumbased systems are common in distributed databases.
- **Fault Tolerance:** Distributed systems are often designed to operate in environments where failures (node crashes, network partitions) are expected. Voting protocols must account for these failures while still achieving consensus.

Common Voting Protocols in Distributed Systems

1. TwoPhase Commit (2PC) Protocol

The TwoPhase Commit protocol is commonly used to ensure atomicity in distributed transactions. It allows multiple participants (nodes) to agree on whether to commit or abort a transaction. It's a blocking protocol, meaning that if one participant fails or becomes unresponsive, the entire system may be delayed.

Phases of 2PC:

Phase 1 (Prepare Phase): The coordinator node sends a `prepare` message to all participating nodes, asking if they are ready to commit. Each participant responds with either a `yes` (if it's ready to commit) or `no` (if it's not ready).

Phase 2 (Commit/Abort Phase): If the coordinator receives `yes` votes from all participants, it sends a `commit` message, and all participants commit the transaction. If any participant responds with a `no` vote (or a failure occurs), the coordinator sends an `abort` message, and all participants roll back their changes.

Pros:

- Simple and straightforward for distributed transactions.
- Ensures atomicity (either all participants commit, or none do).

Cons:

- Blocking: If the coordinator crashes after sending the prepare message, participants may be left waiting indefinitely.
- Cannot tolerate node failures without additional mechanisms.

2. Three Phase Commit (3PC) Protocol

The Three Phase Commit protocol is an improvement on the 2PC protocol, designed to reduce the blocking problem. It introduces an extra phase to minimize the chances of the system waiting indefinitely in case of a failure.

Phases of 3PC:

- Phase 1 (Prepare Phase): Similar to 2PC, the coordinator sends a `prepare` message.
- Phase 2 (Precommit Phase): If all participants are ready, the coordinator sends a `precommit` message, preparing participants for the commit but not actually committing the transaction yet.
- Phase 3 (Commit Phase): The coordinator sends a `commit` message to finalize the transaction.

Pros:

- Nonblocking: Participants can time out and make decisions independently if they don't hear back from the coordinator.
- Improved fault tolerance over 2PC.

Cons:

- More complex than 2PC due to the additional phase.
- Still susceptible to failures in some edge cases, but less blocking than 2PC.

3. Paxos Algorithm

Paxos is one of the most wellknown consensus algorithms in distributed systems, particularly for achieving consensus in unreliable environments where nodes may fail. It is designed to tolerate faults and ensure that a group of nodes can agree on a single value, even if some nodes are down.

Roles in Paxos:

- **Proposers:** Proposers suggest values for the group to agree upon.
- **Acceptors**: Acceptors vote on proposed values and eventually agree on one.
- **Learners**: Once a consensus is reached, learners are informed of the decision.

Phases of Paxos:

1. Prepare Phase: The proposer sends a proposal with a unique ID to a quorum of acceptors, asking them to promise not to accept any earlier proposals.

2. Accept Phase: If the acceptors have not already accepted a higher proposal, they promise to accept the proposal.

3. Commit Phase: If a quorum of acceptors agrees, the proposal is accepted, and the decision is made.

Pros:

- Strong fault tolerance: Paxos can reach consensus even if some nodes fail.
- Does not block: Paxos is designed to make progress as long as a majority of nodes (quorum) are available.

Cons:

- High complexity: Paxos is more difficult to implement and reason about compared to simpler protocols.
- Performance overhead: The messagepassing requirements can lead to performance bottlenecks in large systems.

4. Raft Consensus Algorithm

Raft is a consensus algorithm similar to Paxos but designed to be easier to understand and implement. Raft is used in systems like etcd and HashiCorp Consul for leader election and log replication.

How Raft Works:

Raft divides the problem of consensus into two subproblems: leader election and log replication. A leader is elected among the nodes, and the leader is responsible for managing log entries and ensuring that all other nodes (followers) agree on the same state.

Followers replicate the log entries from the leader, ensuring consistency across the distributed system.

Phases in Raft:

1. Leader Election: If the current leader fails, the followers hold an election to choose a new leader.

2. Log Replication: The leader appends new log entries and replicates them to the followers.

3. Commit: Once a majority of followers confirm the log entries, they are considered committed, ensuring consistency across nodes.

Pros:

- Simpler to understand and implement compared to Paxos.
- Provides strong consistency and fault tolerance.
- Efficient log replication.

Cons:

- The performance is tied to the leader, so a slow leader can affect the overall performance of the system.
- More complex than simpler votingbased protocols like 2PC and 3PC.

5. QuorumBased Voting

In quorumbased voting protocols, nodes vote to reach a decision based on a quorum, which is the minimum number of votes needed to make a decision. This is commonly used in replicated databases to ensure consistency in read and write operations.

Key Operations in QuorumBased Voting:

Read Quorum: The number of nodes that must agree on a read request.

Write Quorum: The number of nodes that must agree on a write request.

For example, in a system with three replicas, a quorum might require two votes to proceed with a read or write operation.

Pros:

- High availability: Allows the system to continue operating as long as a quorum is available.
- Scalable: Suitable for largescale systems with many replicas.

Cons:

- Performance overhead: May require multiple nodes to vote, leading to delays in decisionmaking.
- May not guarantee strong consistency if read and write quorums overlap insufficiently.