UNIT-2

Deadlock Handling Strategies in Distributed System

Deadlocks in distributed systems can severely disrupt operations by halting processes that are waiting for resources held by each other. Effective handling strategies—detection, prevention, avoidance, and recovery—are essential for maintaining system performance and reliability.

Types of Deadlock in Distributed Systems

Below are the types of deadlock in distributed systems:

- **Resource Deadlock**: Processes are stuck waiting for resources held by each other, creating a circular wait.
- **Communication Deadlock**: Processes are waiting for messages from each other that never arrive due to communication issues.
- **Temporal Deadlock**: Deadlocks occur due to timing issues or delays in process execution, leading to a standstill.
- **Deadlock Due to Resource Allocation Policies**: Deadlocks arise from policies or strategies for locking and resource allocation that create cyclic dependencies.
- **Data Deadlock**: Processes are blocked because they are waiting for access to data resources that are locked by each other.

Deadlock Detection in Distributed Systems

Deadlock detection involves identifying when processes in a distributed system are in a deadlock state, where they are blocked because they are waiting for resources held by each other. The primary goal is to recognize the deadlock condition so that corrective actions can be taken.

1. Overview of Detection Methods

Detection methods aim to recognize when a deadlock has occurred by analyzing the system's state. The methods include the following:

- **Resource Allocation Graphs**: Model the relationship between processes and resources to identify deadlocks.
- **Wait-for Graphs**: Simplified version of resource allocation graphs focusing solely on process-to-process relationships.
- **Cycle Detection Algorithms**: Used to find cycles in graphs, indicating deadlocks.

2. Resource Allocation Graphs (RAG)

- **Definition**: A directed graph representing the allocation of resources and the requests for resources among processes.
- **Components**:
    - **Nodes**:
        - **Processes**: Represented by circles or squares.
        - **Resources**: Represented by squares.
    - **Edges**:

        - **Request Edges**: From a process to a resource when a process requests a resource.
        - **Assignment Edges**: From a resource to a process when the resource is allocated to the process.
- **Detection**: Deadlocks are detected by finding cycles in the RAG. A cycle indicates that a group of processes are waiting for resources in a circular manner.

Example: If Process A holds Resource 1 and waits for Resource 2 held by Process B, and Process B waits for Resource 1 held by Process A, a cycle is formed indicating a deadlock.

3. Wait-for Graphs

- **Definition**: A simpler version of the resource allocation graph, focusing on direct process-to-process dependencies.
- **Components**:
    - **Nodes**: Represent processes only.
    - **Edges**:
        - **Wait-for Edges**: From one process to another if the first process is waiting for a resource held by the second process.
- **Detection**: Deadlocks are detected by finding cycles in the wait-for graph. A cycle signifies that processes are waiting on each other, causing a deadlock.

Example: If Process A is waiting for Process B to release a resource, and Process B is waiting for Process A, the wait-for graph will show a cycle.

4. Cycle Detection Algorithms

- **Depth-First Search (DFS)**:
    - **Definition**: A graph traversal algorithm used to detect cycles.
    - **Operation**: DFS explores nodes and edges in a graph. If a back edge (an edge to an ancestor node) is found during traversal, it indicates a cycle.
- **Tarjan's Algorithm**:
    - **Definition**: An efficient algorithm to find strongly connected components (SCCs) in a directed graph.
    - **Operation**: Identifies cycles by finding SCCs, which are subgraphs where every node is reachable from every other node.

Example: Applying DFS to a graph representing resource allocation and request relationships can reveal cycles indicative of deadlock.

Deadlock Prevention in Distributed Systems

Deadlock prevention involves designing a system in such a way that the conditions for deadlock are never met. This approach involves modifying the system's behavior to avoid circular wait conditions.

1. Basic Principles of Deadlock Prevention

- **Avoiding Conditions**: The main goal is to prevent the four necessary conditions for deadlock (Mutual Exclusion, Hold and Wait, No Preemption, Circular Wait) from occurring.

2. Resource Allocation Policies

- **Policy Design**: Establish policies that prevent circular wait conditions. This might involve ordering resources or defining rules for resource allocation.

Example*: Allocate resources in a specific order to prevent circular waits. For example, if resources are ordered as R1, R2, R3, processes must request resources in this order.*

3. Hold and Wait Prevention

- **Strategy**: Processes must request all resources they need at once, rather than holding some resources while waiting for others.

Example*: A process must request all required resources before it starts execution, thus avoiding holding resources while waiting for additional ones.*

4. Preemption Strategies

- **Definition**: Allow resources to be taken away from processes and reallocated to others to break a deadlock.
- **Operation**: Resources held by a process involved in deadlock are preempted and allocated to other processes.

Example*: If a process is holding a resource and waiting for another, preempting the held resource and reallocating it can help break the deadlock.*

5. Request Ordering

- **Definition**: Enforce an ordering of resource requests to avoid circular waits.
- **Operation**: Require processes to request resources in a predefined order, ensuring that no circular wait conditions can form.

Example*: If resources are ordered as R1, R2, R3, processes must request R1 before R2 and R2 before R3.*

Deadlock Avoidance in Distributed Systems

Deadlock avoidance involves designing systems to dynamically allocate resources in a way that avoids deadlock by ensuring that every resource request is granted only if it leaves the system in a safe state.

1. Banker's Algorithm

- **Definition**: An algorithm for allocating resources in a way that avoids deadlock by ensuring that the system remains in a safe state.
- **Components**:
    - **Available Matrix**: Shows available resources.
    - **Allocation Matrix**: Shows currently allocated resources.
    - **Request Matrix**: Indicates requested resources.
- **Operation**: When a process requests resources, the system checks if granting the request keeps the system in a safe state. If so, the request is granted; otherwise, it is denied.

Example*: If a process requests resources and the system can still guarantee that all other processes can complete with the remaining resources, the request is granted.*

2. Safe and Unsafe States

- **Safe State**: A state where there exists at least one sequence of processes that can complete without causing a deadlock.
- **Unsafe State**: A state where no such sequence exists, potentially leading to a deadlock if resource requests are granted.

Example*: In a safe state, the system can allocate resources in such a way that all processes eventually complete. In an unsafe state, resource allocation might lead to deadlock.*

3. Resource Allocation Policies

- **Policies**: Implement policies that ensure resource requests do not lead to unsafe states. This involves dynamic checks based on the Banker's Algorithm or similar approaches.

Example*: Policies to ensure that resource requests are evaluated dynamically to maintain the system's safety, preventing transitions into unsafe states.*

4. Dynamic Approaches to Deadlock Avoidance

- **Dynamic Resource Allocation**: Continuously monitor and adjust resource allocations to prevent unsafe states.
- **Operation**: Adjust resource allocations dynamically based on the current state and resource requests to ensure the system remains in a safe state.

Example*: Continuously applying deadlock avoidance checks and adjusting resource allocations to adapt to changing demands and system states.*

Deadlock Recovery in Distributed Systems

Deadlock recovery involves taking corrective actions to resolve deadlocks once they are detected, restoring normal system operations.

1. Recovery Strategies Overview

- **Strategies**: Approaches to resolve deadlocks typically include terminating processes, preempting resources, or rolling back transactions.

2. Process Termination

- **Definition**: Kill one or more processes involved in the deadlock to break the circular wait.
- **Operation**: Select processes to terminate based on criteria such as least cost or impact on the system.

Example*: Terminating the processes involved in a deadlock, starting with the least important or least costly to terminate.*

3. Resource Preemption

- **Definition**: Temporarily take resources away from some processes and reallocate them to others to break the deadlock.
- **Operation**: Preempt resources from processes involved in the deadlock and reallocate them to other processes to break the cycle.

***Example**: If Process A and B are in a deadlock, preempt resources from one and allocate them to the other to resolve the deadlock.*

4. Rollback and Restart

- **Definition**: Roll back processes to a previous state and restart them to avoid deadlock conditions.
- **Operation**: Use checkpoints or logs to revert processes to a state before they entered the deadlock and then restart them.

***Example**: Rolling back a transaction to a checkpoint before the deadlock occurred and restarting it from that point.*

5. Choosing a Recovery Strategy

- **Criteria**: Consider factors such as system overhead, performance impact, and the nature of the deadlock when choosing a recovery strategy.

***Example**: Select a strategy based on the specific context of the deadlock, balancing the cost of recovery with the impact on system performance and reliability.*

By understanding and applying these concepts, distributed systems can effectively manage and mitigate the impact of deadlocks, ensuring smooth operation and reliable performance.

Issues of Deadlock Detection

Various issues of deadlock detection in the distributed system are as follows:

1. Deadlock detection-based deadlock handling requires addressing two fundamental issues: first, detecting existing deadlocks, and second, resolving detected deadlocks.
2. Detecting deadlocks entails tackling two issues: WFG maintenance and searching the WFG for the presence of cycles.
3. In a distributed system, a cycle may include multiple sites. The search for cycles is highly dependent on the system's WFG as represented across the system.

Control Organization for Distributed Deadlock Detection Algorithms

Algorithms for detecting distributed deadlock can be handled in three different ways:

- Centralized
- Distributed
- Hierarchical

Assume that the network supports reliable communication.

Centralized:

One central site sets up a global WFG and searches for cycles.

All decisions are made by the central control node.

- It must maintain the global WFG constantly **or**
- Periodically reconstruct it.

The main advantage is that this permits the use of relatively simple algorithms.

The disadvantages include the following:

- There is one, single point of failure.
- There can be a communication bottleneck around the site due to all the WFG information messages.
- Furthermore, this traffic is independent of the formation of any deadlock.

Distributed:

In a distributed control organization,

- All sites have an equal amount of information.
- All sites make decisions based on local information.
- All sites bear equal responsibility for the final decision in detecting deadlock.
- All sites expend equal effort to the final decision.
- The global WFG is spread across the sites.
- Deadlock detection is initiated whenever a process thinks there might be a problem.
- Several sites can initiate the detection at the same time.

The advantages include the following:

- There is no central point of failure.
- A single node failure cannot cause a crash.
- There is no *one* site with heavy traffic due to the detection algorithm.
- The algorithm is only initiated when process(es) feel there might be a problem.
- The algorithm is not run periodically, only when needed.

The main disadvantage is that resolution may be difficult, as not all sites may be aware of the processes involved in the deadlock. The proof of correctness for this type of algorithm may be difficult.

Hierarchical:

The sites (nodes) are logically connected in a hierarchical structure (such as a tree).

A site can detect deadlock in its descendants. This type of algorithm has the best of both the centralized and the distributed deadlock detection algorithms. For efficiency purposes, it is best to keep clusters of interacting processes together in the hierarchy.

Agreement Protocols in Distributed Systems

Agreement protocols in distributed systems ensure that multiple nodes or processes reach a consensus on a shared state or decision despite failures and network partitions. This introduction explores the fundamental concepts, challenges, and key protocols used to achieve reliable agreement in decentralized environments.

These protocols are crucial for maintaining consistency and reliability in decentralized environments. Key types include:

- **Consensus Protocols:** Ensure all nodes agree on a single value or decision (e.g., Paxos, Raft).
- **Atomic Broadcast:** Guarantees that messages are delivered to all nodes in the same order (e.g., Total Order Broadcast).
- **Two-Phase Commit (2PC):** A protocol for ensuring all participants in a distributed transaction agree to commit or abort the transaction.

Agreement protocols handle challenges such as network failures, process crashes, and message delays, ensuring the system operates reliably and consistently.

Importance of Agreement Protocols in Distributed Systems

Agreement protocols are crucial in distributed systems for several reasons:

- **Consistency:** They ensure all nodes or processes agree on a single state or decision, maintaining consistency across the system despite failures or network partitions.
- **Fault Tolerance:** By enabling consensus even when some components fail, agreement protocols enhance the system's ability to recover and continue operating.
- **Coordination:** They facilitate coordinated actions and resource management among distributed nodes, crucial for tasks like distributed transactions and coordination of tasks.
- **Reliability:** Agreement protocols help ensure that distributed systems function correctly and consistently, which is essential for critical applications like financial transactions and cloud services.
- **Scalability:** They support scalable and reliable communication and operation as the system grows and more nodes are added.

Types of Agreement Protocols in Distributed Systems

Agreement protocols in distributed systems are designed to ensure that all participating nodes or processes agree on a single decision or state despite failures and network issues. Key types include:

1. Consensus Protocols

- **Paxos:** Achieves consensus by ensuring that a majority of nodes agree on a proposed value. It deals with node failures and network partitions but can be complex to implement.
- **Raft:** A more understandable consensus protocol compared to Paxos, Raft uses a leader-follower model to manage log replication and ensure that all nodes agree on the state changes.

2. Two-Phase Commit (2PC)

- **Basic Two-Phase Commit (2PC):** A protocol used to coordinate a distributed transaction by having a coordinator node ask all participant nodes to either commit or abort the

transaction. It ensures all nodes agree on the transaction outcome but can be vulnerable to blocking if participants or coordinators fail.

3. Three-Phase Commit (3PC)

- **Three-Phase Commit (3PC):** An extension of 2PC that adds an additional phase to reduce the likelihood of blocking. It introduces a pre-commit phase to improve fault tolerance and ensure that the protocol can handle coordinator failures more gracefully.

4. Atomic Broadcast Protocols

- **Total Order Broadcast:** Guarantees that all messages are delivered to all nodes in the same order. This protocol ensures that even if nodes receive messages out of order, they will process them in a consistent sequence.

5. Byzantine Fault Tolerance (BFT) Protocols

- **Practical Byzantine Fault Tolerance (PBFT):** Designed to handle Byzantine failures where nodes may act maliciously or arbitrarily. PBFT achieves consensus by requiring a majority of nodes to agree on a decision, assuming some nodes might be faulty or malicious.

Each type of protocol addresses specific challenges related to achieving consensus in distributed environments, ranging from simple fail-stop failures to more complex Byzantine failures.

Classical Agreement Protocols in Distributed Systems

Classical agreement protocols in distributed systems are fundamental mechanisms designed to ensure consistency and consensus among distributed nodes or processes. Here are some of the most well-known classical protocols:

1. Paxos

Paxos is a consensus algorithm that enables a group of distributed nodes to agree on a single value, even if some nodes fail. It is designed to handle node crashes and network partitions.

- **Key Components:**
  - **Proposers:** Propose values to be agreed upon.
  - **Acceptors:** Accept proposals and decide on a value.
  - **Learners:** Learn the value chosen by the majority of acceptors.
- **Phases:**
  - **Prepare:** A proposer requests to become a leader by sending a prepare request to acceptors.
  - **Promise:** Acceptors respond with a promise not to accept lower-numbered proposals and provide information about any previously accepted proposals.
  - **Propose:** The proposer sends a proposal to the acceptors, who then decide if the proposal is accepted.
- **Strengths:** Ensures consensus despite failures of some nodes; widely studied and implemented.
- **Weaknesses:** Complex to implement and understand; can face performance issues with large numbers of nodes.

2. Two-Phase Commit (2PC)

2PC is a protocol used to coordinate distributed transactions to ensure that all participants either commit or abort the transaction. It is simpler compared to Paxos but can suffer from blocking issues.

- **Phases:**

    - **Prepare Phase:** The coordinator node asks all participant nodes to prepare for the transaction and vote on whether to commit or abort.
    - **Commit Phase:** Based on votes, the coordinator instructs all participants to either commit the transaction if all agree or abort if any participant votes to abort.
- **Strengths:** Simplicity in coordination and implementation; guarantees atomicity of transactions.
- **Weaknesses:** Blocking can occur if the coordinator or participants fail; no progress if some participants do not respond.

3. Three-Phase Commit (3PC)

3PC is an enhancement of 2PC designed to reduce the likelihood of blocking. It adds an additional phase to improve fault tolerance.

- **Phases:**
    - **Can Commit Phase:** The coordinator asks participants if they are able to commit. Participants respond with either "Yes" or "No."
    - **Pre-Commit Phase:** If all participants reply "Yes," the coordinator sends a pre-commit message. Participants then prepare to commit.
    - **Do Commit Phase:** The coordinator sends a commit message if all participants have acknowledged the pre-commit phase. Participants commit the transaction.
- **Strengths:** Reduces blocking compared to 2PC; handles some failures more gracefully.
- **Weaknesses:** More complex than 2PC; still vulnerable to certain types of failures.

4. Atomic Broadcast (Total Order Broadcast)

Atomic Broadcast ensures that all messages are delivered to all nodes in the same order. It is crucial for maintaining consistency in systems where message ordering is important.

- **Key Concepts:**
    - **Total Order:** All nodes see the messages in the same order.
    - **Atomicity:** Guarantees that either all nodes receive the message or none do.
- **Strengths:** Ensures consistent message ordering across distributed nodes.
- **Weaknesses:** Can be complex to implement; may introduce additional latency.

5. <u>Byzantine Fault Tolerance (BFT) Protocols</u>

BFT protocols handle cases where nodes may act maliciously or arbitrarily (Byzantine failures). They ensure consensus despite nodes exhibiting faulty or adversarial behavior. Example: Practical Byzantine Fault Tolerance (PBFT):

- **Phases:**
    - **Pre-Prepare:** The primary node proposes a value.
    - **Prepare:** Nodes validate the proposal and broadcast their votes.
    - **Commit:** Nodes finalize the decision based on the majority agreement.
- **Strengths:** Handles arbitrary and malicious failures; ensures consensus in adversarial environments.
- **Weaknesses:** Higher overhead and complexity compared to non-Byzantine protocols.

These classical agreement protocols form the basis for many distributed systems' consensus and coordination mechanisms, addressing various challenges related to consistency, reliability, and fault tolerance

Modern Agreement Protocols in Distributed Systems

Modern agreement protocols in distributed systems build on classical protocols to address new challenges, improve performance, and handle larger-scale or more complex environments. Here are some prominent modern agreement protocols:

1. Raft

Raft is a consensus algorithm designed to be more understandable and practical than Paxos. It uses a leader-based approach to manage log replication and ensure that all nodes in the system agree on the order of operations.

- **Key Components:**
  - **Leader:** A node that manages the replication of logs and handles client requests.
  - **Followers:** Nodes that replicate the leader's log entries.
  - **Candidates:** Nodes that seek to become the leader during elections.
- **Phases:**
  - **Leader Election:** Nodes elect a leader to manage the consensus process.
  - **Log Replication:** The leader replicates log entries to followers.
  - **Safety and Commitment:** Once a majority of nodes replicate an entry, it is considered committed.
- **Strengths:** Simplifies the consensus process with a clear leader and robust mechanisms for leader election and log replication.
- **Weaknesses:** The leader can become a bottleneck and single point of failure.

2. Tendermint

Tendermint is a Byzantine Fault Tolerant (BFT) consensus algorithm designed for high-performance blockchains and distributed applications. It achieves consensus despite nodes exhibiting arbitrary or malicious behavior.

- **Key Components:**
  - **Proposers:** Propose new blocks or transactions.
  - **Validators:** Validate proposed blocks and vote on them.
  - **BFT Protocol:** Ensures consensus by requiring a supermajority of validators to agree.
- **Phases:**
  - **Proposal:** A proposer suggests a new block.
  - **Pre-vote and Pre-commit:** Validators vote on the proposal.
  - **Commit:** A block is committed if it receives a supermajority of votes.
- **Strengths:** Provides high throughput and fast finality while tolerating malicious nodes.
- **Weaknesses:** Requires a supermajority of nodes to be honest or fault-free to achieve consensus.

3. Practical Byzantine Fault Tolerance (PBFT)

PBFT is designed to tolerate Byzantine failures, where nodes may act arbitrarily or maliciously. It ensures that all non-faulty nodes reach agreement despite a fraction of nodes potentially being faulty.

- **Key Components:**
  - **Primary:** Proposes new requests or transactions.
  - **Replicas:** Validate and respond to requests.
  - **View Changes:** Mechanism for changing the primary if it fails.
- **Phases:**
  - **Pre-prepare:** The primary node proposes a request.

    - **Prepare:** Replicas broadcast their agreement on the proposal.
    - **Commit:** Replicas reach a consensus and execute the request.
- **Strengths:** Ensures robustness in adversarial environments and provides strong consistency.
- **Weaknesses:** High communication overhead and complexity due to the need for multiple phases and message exchanges.

Consensus in Blockchain Systems

Below is how consensus work in blockchain systems:

1. Proof of Work (PoW)

PoW requires participants (miners) to solve complex cryptographic puzzles to validate transactions and create new blocks. The first miner to solve the puzzle gets to add the block to the blockchain and is rewarded.

Example: *Bitcoin uses PoW as its consensus mechanism.*

2. Proof of Stake (PoS)

PoS selects validators based on the number of coins they hold and are willing to "stake" as collateral. Validators are chosen to create and validate blocks based on their stake and other factors like randomization.

Example: *Ethereum plans to transition from PoW to PoS with Ethereum 2.0.*

3. Delegated Proof of Stake (DPoS)

DPoS involves stakeholders voting for a small number of delegates who validate transactions and create blocks on their behalf. This creates a more efficient but less decentralized model.

Example: *EOS and TRON use DPoS for consensus.*

4. Practical Byzantine Fault Tolerance (PBFT)

PBFT is designed to handle Byzantine faults where nodes may act arbitrarily or maliciously. It requires a majority of nodes to agree on the validity of transactions and state changes.

Example: *Hyperledger Fabric uses PBFT and similar variants in its consensus algorithms.*

5. Proof of Authority (PoA)

PoA relies on a small number of trusted nodes (authorities) to validate transactions and create blocks. These nodes are pre-approved and known to be trustworthy.

Example: *VeChain and certain instances of Ethereum-based private networks use PoA.*

Challenges of Agreement Protocols in Distributed Systems

Key challenges of agreement protocols in distributed systems, summarized:

- **Fault Tolerance:** Handling node failures and maintaining consensus despite crashes or faulty behavior.
- **Network Partitions:** Managing split-brain scenarios and reconciling disconnected nodes when connectivity is restored.
- **Scalability:** Ensuring performance remains efficient as the number of nodes increases.
- **Latency and Throughput:** Balancing the time to reach consensus with the system's transaction processing capacity.
- **Complexity:** Dealing with the complexity of implementation and ensuring correctness.
- **Security:** Protecting against malicious attacks and Byzantine failures.
- **Leader Election:** Efficiently electing and managing a leader node and handling leader failures.