**Self-Stabilization in Distributed System:Definition of self-stabilization, Issues in the design of self-stabilization algorithms, Self-stabilizing distributed spanning trees, Self-stabilizing algorithms for spanning-tree construction:Dolev, Israeli, and Moran algorithm.**

**Self-Stabilization in Distributed Systems**

**1. Definition of Self-Stabilization**

* **Self-Stabilization** is a property of a distributed system that guarantees it will **eventually converge to a correct behavior**, starting from any arbitrary state, **without external intervention**.
* First introduced by **Edsger W. Dijkstra** in 1974.

**Key Properties:**

* **Convergence:** The system must reach a legitimate state after a finite number of steps.
* **Closure:** Once the system reaches a legitimate state, it must stay there unless faults occur.

**2. Issues in the Design of Self-Stabilizing Algorithms**

1. **Arbitrary Initial State:**
   * System must work correctly from any possible configuration.
2. **Fault Model:**
   * Transient faults (e.g., memory corruption, message loss) may lead to incorrect states.
3. **Lack of Synchrony:**
   * Algorithms must work under asynchronous settings (no global clock).
4. **No Initialization Assumptions:**
   * Processes may not have global knowledge or consistent views.
5. **Fairness:**
   * Algorithms must assume **fair scheduling** (every process eventually gets to execute).
6. **Termination Guarantee:**
   * Algorithm must terminate in a finite number of steps regardless of the initial state.
7. **Scalability and Complexity:**
   * The time and message complexity should be acceptable for large networks.

**3. Self-Stabilizing Distributed Spanning Trees**

**Objective:**

To construct a **spanning tree** rooted at a particular node in a self-stabilizing manner — even if the network starts in a faulty or inconsistent state.

**Requirements:**

* No cycles in the tree.
* All nodes must be connected.
* A unique root is selected.
* Eventually converges to a correct tree structure.

**4. Dolev, Israeli, and Moran (DIM) Algorithm**

**Overview:**

* A **self-stabilizing algorithm** for constructing a spanning tree in a distributed system.
* Proposed by **Shlomi Dolev, Amos Israeli, and Shlomo Moran**.
* Works under **asynchronous** message-passing systems.
* Builds a rooted spanning tree from **any initial state**.

**Algorithm Idea:**

Each node maintains:

* parent: pointer to its parent in the tree.
* level: distance from the root node.

Each node periodically:

* Exchanges information with neighbors.
* Chooses as its parent the neighbor with the **minimum level**.
* Updates its level to parent.level + 1.

**Rules:**

1. If the node is the **root**, its level = 0, parent = null.
2. If a node finds a neighbor with a **lower level**, it updates:

parent = neighbor

level = neighbor.level + 1

1. If no change is required, the node does nothing.

**Convergence:**

* After a finite number of steps, all nodes point to a parent with a lower level.
* Eventually, a **tree rooted at the root node** is formed.
* Cycles and inconsistencies are eliminated automatically.

**Features:**

* Self-stabilizing: works from any arbitrary state.
* Handles transient faults.
* No need for global knowledge or synchronization.

**Summary**

| **Concept** | **Description** |
| --- | --- |
| Self-Stabilization | System corrects itself from any initial state |
| Convergence | Reaches a valid state in finite time |
| Closure | Stays in correct state unless faults occur |
| DIM Algorithm | Constructs self-stabilizing spanning tree |
| Key Features | Asynchronous, fault-tolerant, distributed |

**Consistency and Replication: Data-centric consistency models, Client-centric consistency models, Consistency protocols.**

* **Data-Centric Consistency Models**
* **Client-Centric Consistency Models**
* **Consistency Protocols**

**Consistency and Replication in Distributed Systems:**

**Introduction to Replication**

* Replication means keeping multiple copies of the same data on different nodes.
* Helps improve:
  + Availability
  + Fault tolerance
  + Scalability
  + Performance

However, maintaining consistency among replicas is challenging.

**2. Data-Centric Consistency Models**

These define the behavior of read/write operations from the perspective of the entire system.

a) Strict Consistency

* Any read returns the most recent write.
* Implies global time — difficult to implement in distributed systems.

b) Linearizability (Atomic Consistency)

* Operations appear to happen instantaneously at some point between invocation and response.
* Realistic version of strict consistency with respect to real-time ordering.

c) Sequential Consistency

* All processes see the same sequence of operations.
* No need for real-time ordering, but all reads/writes must appear in same global order.

d) Causal Consistency

* Writes that are causally related must be seen by all in the same order.
* Concurrent (unrelated) writes may be seen in different orders.

e) FIFO (PRAM) Consistency

* Writes from a single process are seen in order, but no guarantees for writes from different processes.

f) Eventual Consistency

* All replicas will converge to the same value eventually if no updates occur.
* Common in large-scale systems like Amazon Dynamo, Cassandra**.**

**3. Client-Centric Consistency Models**

These models relax consistency guarantees from the client's point of view. Useful when:

* Clients access different replicas over time.
* Network partitions are possible.

**a) Monotonic Reads**

* Once a client reads a value, all future reads return the same or newer values.

b) Monotonic Writes

* A client’s writes are applied in the same order as they were issued.

**c) Read Your Writes**

* A client always sees its own updates in subsequent reads.

**d) Writes Follow Reads**

* A write following a read is guaranteed to be applied to a replica that has seen that read.

**4. Consistency Protocols**

**These are mechanisms used to enforce a chosen consistency model across replicas.**

**a) Primary-Based Protocols**

* One replica is designated as primary.
* All writes go to primary; it updates other replicas.

**i) Remote-Write Protocol:**

* All writes and reads go through the primary.

**ii) Local-Write Protocol:**

* Writes are done locally, then forwarded to the primary for propagation.

**b) Quorum-Based Protocols**

* Use quorums (read/write majority voting) to ensure consistency.
* **Ensure:**

W + R > N, where W = write quorum, R = read quorum, N = total replicas.

**c) Replication Using Gossip Protocols**

* Nodes periodically gossip with others to exchange updates.
* Ensures eventual consistency with low overhead.

**d) Cache Coherence Protocols (for memory systems)**

* Used in distributed shared memory systems.
* Examples: Write-invalidate, write-update protocols.

**Fault-Toleranence: Introduction to fault tolerance, Process resilience, Distributed commit. Checkpointing and rollback recovery . Agreement in (message-passing) synchronous systems with failures, Agreement in asynchronous message-passing systems with failures.**

**Fault Tolerance in Distributed Systems**

**. Introduction to Fault Tolerance**

* Fault Tolerance is the ability of a system to continue operating correctly even if some of its components fail.
* Essential in distributed systems where failures are inevitable due to network issues, hardware faults, or crashes.

**Types of Faults:**

* Crash faults: A process stops functioning.
* Omission faults: Messages are lost or not sent.
* Timing faults: Delays beyond allowed limits.
* Byzantine faults: Arbitrary or malicious failures.

**2. Process Resilience**

* A system is k-resilient if it can tolerate up to k process failures.
* Techniques:
  + Replication: Duplicate processes or data across multiple nodes.
  + Voting: Use majority decisions (e.g., Paxos, Raft).
  + Failure detectors: Monitor and detect crashed nodes.

**3. Distributed Commit**

Ensures atomicity in distributed transactions (all-or-nothing execution).

a) Two-Phase Commit (2PC)

1. Phase 1: Prepare
   * Coordinator asks participants to prepare.
   * Participants reply with “Yes” or “No”.
2. **Phase 2: Commit/Abort**
   * If all reply “Yes”: Commit
   * If any say “No”: Abort

**Drawback: Blocking —** if the coordinator crashes, participants may wait forever.

b) Three-Phase Commit (3PC)

* Adds an extra phase to avoid blocking.
* Non-blocking, but assumes no network partition and uses timeouts.

**4. Checkpointing and Rollback Recovery**

**Used to restore a system to a consistent state after a fault.**

**a) Checkpointing**

* **Save a snapshot of the process's state periodically.**
* **Types:**
  + Uncoordinated Checkpointing: Each process checkpoints independently — may cause domino effect.
  + Coordinated Checkpointing: All processes checkpoint together to avoid inconsistencies.
  + Communication-Induced Checkpointing: Induced based on message patterns**.**

**b) Rollback Recovery**

* On failure, process rolls back to last checkpoint.
* Log-based recovery may also replay messages or actions after the checkpoint.

**5. Agreement in Synchronous Message-Passing Systems with Failures**

**Problem**: Consensus/Agreement

* All non-faulty processes must agree on the same value, even in the presence of failures.

Model:

* Synchronous system (bounded message delays, global clock).
* Up to f processes may crash.

Algorithm:

* Consensus is possible if f < n, where n = total processes.
* Many rounds of message exchange used to mask failures.

6. Agreement in Asynchronous Message-Passing Systems with Failures

FLP Impossibility Result:

* Fischer, Lynch, and Paterson (FLP) proved that:

*No deterministic consensus algorithm can guarantee termination in an asynchronous system with even one crash failure.*

Solutions (By relaxing assumptions):

* Randomized algorithms (e.g., Ben-Or's algorithm).
* Failure detectors (e.g., Chandra-Toueg model).
* Quorum-based protocols.
* Paxos Algorithm (practical, tolerates crash faults).
* Raft Algorithm (easier-to-understand alternative to Paxos).