Module 5

Q. RAFT Consensus Algorithm

=>

1. The **RAFT Consensus Algorithm** is a protocol used to maintain **consistency across distributed systems**.
2. It ensures that even when multiple computers (called **nodes**) are working together, they **all agree on the same data or state**.
3. RAFT is mainly used in systems where **data replication** is required, such as **blockchains, databases, or cluster systems**.
4. The algorithm was designed by **Diego Ongaro and John Ousterhout in 2013** as a simpler alternative to the **Paxos algorithm**.
5. RAFT is easy to understand because it divides the consensus process into **three clear roles** and **three main stages**.

Main Goal of RAFT

6. The main goal of the RAFT algorithm is to **achieve consensus on a shared log of operations** across multiple servers.
7. This means every node in the network should have the **same sequence of commands or transactions**, even if some nodes fail.
8. Example: In a distributed banking system, if one server records a transaction "Deposit ₹1000," then all servers must record it in the same order.
9. RAFT guarantees that once a transaction is committed, **it will never be lost**, even if a node crashes or restarts.

Three Main Stages of RAFT

1. Leader Election

1. The process begins when all nodes start as **followers**.
2. If a follower does not receive any message (called a heartbeat) from a leader within a **timeout period**, it becomes a **candidate**.
3. The candidate then **requests votes** from all other nodes to become the leader.
4. Each node can vote for only one candidate per term.
5. If a candidate receives **a majority of votes**, it becomes the **leader** for that term.
6. Example: In a 5-node cluster, if Node 2 gets 3 votes, it becomes the leader.
7. The leader then sends periodic **heartbeat messages** to followers to confirm that it is still active.

2. Log Replication

1. Once the leader is elected, all client requests are sent only to the leader.
2. The leader appends each client request to its **log** and assigns it a unique **index number**.
3. Then, it sends these log entries to all followers in the network.
4. Each follower stores the entry temporarily and sends an acknowledgment back to the leader.
5. When the leader receives acknowledgments from a **majority of followers**, it marks the entry as **committed**.
6. Finally, the leader notifies all followers to **commit the same entry** to their state machines.
7. This ensures that all nodes have **identical logs** and are in sync.
8. Example: If a transaction “Transfer ₹500 from A to B” is logged by the leader, it will be replicated to all followers in the same order.

3. Safety and Log Consistency

1. RAFT ensures that **no two leaders can exist simultaneously** in the same term.
2. This is achieved using **term numbers**, which increase each time a new election occurs.
3. Followers always accept logs only from the **current leader** with the highest term number.
4. Even if a leader crashes, a new one will be elected quickly, ensuring **high availability**.
5. If a follower’s log becomes inconsistent, the leader will **overwrite incorrect entries** to maintain uniformity.
6. Example: If Node 5 missed a transaction during downtime, the leader will resend the missing logs once it rejoins the network.

Example of RAFT Working (Simplified Scenario)

1. Suppose there are **five nodes**: N1, N2, N3, N4, and N5.
2. Initially, all nodes are followers and waiting for heartbeats.
3. After the timeout, N3 becomes a candidate and requests votes from others.
4. N1, N2, and N3 vote for N3, so it becomes the **leader**.
5. Now, a client sends a command: “Add ₹1000 to Account X” to the leader (N3).
6. N3 adds this command to its log and sends it to N1, N2, N4, and N5.
7. Once N3 gets confirmations from at least three nodes (majority), it marks the entry as **committed**.
8. Then all nodes update their local state machines with the new value.
9. If N3 fails, the remaining nodes will hold a new election, and a new leader (say N2) will take over automatically.

Q. PAXOS Consensus Algorithm

=>

1. The **PAXOS Consensus Algorithm** is a method used in distributed systems to make sure that **multiple nodes agree on a single value**, even if some of them fail.
2. It was developed by **Leslie Lamport** in the late 1980s and is considered one of the most important algorithms for **distributed consensus**.
3. The main goal of Paxos is to ensure that a **group of nodes (or servers)** can **agree on one decision** in a **fault-tolerant** manner.
4. It is used in many real-world systems such as **Google Chubby**, **Microsoft Azure**, and **Apache ZooKeeper**.
5. Paxos is designed for systems where **nodes can crash, messages can be delayed, or duplicated**, but the system should still continue to function correctly.

Main Idea of Paxos

1. Paxos ensures that all nodes in a distributed network **agree on a single value (for example, a transaction or block)**.
2. Once a value is agreed upon, it becomes **final and cannot be changed**.
3. The algorithm works even if some nodes fail, as long as a **majority of nodes are still active**.
4. Example: If a distributed database must decide which transaction to apply first, Paxos ensures that **all servers agree on the same order**.
5. This agreement avoids conflicts and keeps data **consistent** across all nodes.

Phases of Paxos Algorithm

Paxos works in **two main phases**: the **Prepare Phase** and the **Accept Phase**.

1. Prepare Phase

1. The **Proposer** selects a proposal number n (a unique number greater than any previous proposal number).
2. It sends a **prepare request** to all **Acceptors** asking if they can promise not to accept any proposal with a number less than n.
3. Each **Acceptor** compares the received proposal number with its previously accepted one.
4. If n is higher than any previous number, the Acceptor **promises not to accept any smaller proposal** and sends back the highest proposal it has already accepted (if any).
5. Example: If the proposer sends proposal number 5, and it is higher than previous proposals, the acceptors promise not to accept proposals numbered 4 or lower.

2. Accept Phase

1. After receiving a majority of promises, the proposer sends an **accept request** with a value v and the same proposal number n to all Acceptors.
2. Each **Acceptor** accepts this proposal if it has not already promised a higher proposal number.
3. Once a majority of Acceptors accept the proposal, the value is **chosen** (consensus is reached).
4. Example: If 3 out of 5 Acceptors accept proposal number 5 with value “Transaction A,” then “Transaction A” becomes the agreed value.
5. After that, all **Learners** are informed of the chosen value, and they update their local state.

Example of Paxos in Action

1. Imagine a distributed system with **five servers (A, B, C, D, E)**.
2. Server A wants to propose the value **“X”** for the next transaction.
3. Server A sends a **prepare message** with proposal number 10 to all servers.
4. Servers B, C, D, and E respond with promises since 10 is the highest number they’ve seen.
5. Server A now sends an **accept request** for value “X” with proposal number 10.
6. If three or more servers accept it (majority), “X” becomes the **agreed value**.
7. All learners then update their records, and the system agrees that “X” is the final decision.
8. If Server A fails midway, another proposer (say Server B) can start a new proposal with a higher number (like 11) to continue the process.

Q. State Machine Replication (SMR)

=>

- State Machine Replication (SMR) is a method used in distributed systems to ensure that multiple computers or nodes perform the same actions in the same order.
- It helps maintain **consistency and reliability** even if some nodes fail or act maliciously.
- The main goal is to make the system behave as if there is only a single machine, even though multiple replicas exist.
- Each node in the system maintains a copy of the same state and applies the same sequence of operations.
- These operations are usually commands or transactions sent by clients to update the system's state.
- The **"state machine"** means that given a specific state and a command, the system always produces the same output and new state.
- SMR ensures **fault tolerance**, meaning that the system continues to work correctly even if some servers go down.

Workflow of State Machine Replication

- The system has multiple replicas (or nodes), each maintaining the same application logic and state.
- A client sends a transaction or command (e.g., "Add ₹500 to account A") to one of the nodes.
- The nodes communicate with each other to **reach consensus** on the order of commands.
- After consensus is reached, each node executes the command in the same order.
- As all nodes execute the same commands, they all reach the same resulting state.
- The client then receives a response confirming that the operation was successfully performed.
- This process repeats for every client request to ensure consistency across the network.

State Machine Replication in Crowdfunding (Example)

- In a **blockchain-based crowdfunding system**, multiple nodes maintain a record of contributions and funding goals.
- When a contributor donates ₹1000 to a project, this action must be recorded consistently across all nodes.
- The SMR ensures that every node applies the "Add ₹1000 to Project A" transaction in the same order.
- Even if one node fails, others can continue processing transactions while maintaining the same state.

- Once consensus is achieved, the new total funding amount is identical across all nodes.
- This prevents inconsistencies like one node showing ₹10,000 while another shows ₹11,000.

Q. Hyperledger Fabric v1 Architecture

=>

- **Hyperledger Fabric** is a **permissioned blockchain framework** developed by the Linux Foundation for enterprise use.
- It allows multiple organizations to work together securely while maintaining privacy and trust.
- Unlike public blockchains such as Ethereum or Bitcoin, Fabric is designed for **business networks** where all participants are known and verified.
- It follows a **modular architecture**, which means components like consensus, membership, and ordering can be customized.
- Fabric v1 introduced a **new architecture** that separates transaction execution, ordering, and validation for better scalability and performance.
- This separation is known as the **"execute–order–validate"** model.
- It replaced the older **"order–execute"** model used in traditional blockchains to avoid problems like nondeterministic execution.

Main Components of Hyperledger Fabric v1

1. Peers

- Peers are the main nodes in the Fabric network that maintain the **ledger** and **state database**.
- Every peer stores a copy of the blockchain ledger and executes smart contracts (called chaincode).
- Peers can have different roles such as **endorsing peers**, **committing peers**, or **anchor peers**.
- Example: In a supply chain network, Company A and Company B may each have their own peer node to maintain data.

2. Orderer (Ordering Service)

- The **Ordering Service** is responsible for arranging transactions into a specific order.
- It collects transactions from different peers and creates **blocks** in a chronological sequence.
- The ordering service is independent of transaction execution, which improves efficiency.
- It can use different consensus mechanisms like **Solo**, **Kafka**, or **Raft**.
- Example: In a banking consortium, the ordering service ensures all banks record transactions in the same order to prevent double-spending.

3. Chaincode (Smart Contracts)

- Chaincode is the name given to **smart contracts** in Hyperledger Fabric.
- It defines the **business logic** that executes transactions between network participants.
- Chaincode runs in a **Docker container** separate from the main peer process for security.
- Example: A chaincode might define rules such as "transfer ownership of goods after payment confirmation."

4. Channels

- A **channel** is a private communication layer that allows a group of organizations to have a **separate ledger**.
- Each channel has its own blockchain, shared only among its members.
- This provides **data privacy** and **confidentiality** in multi-organization environments.
- Example: In a supply chain, the manufacturer and supplier can share one channel while the retailer uses another channel.

5. Membership Service Provider (MSP)

- MSP handles **identity management** and **authentication** of participants in the network.
- It ensures that only authorized users and nodes can perform operations.
- The MSP issues **digital certificates** through a **Certificate Authority (CA)**.
- Example: A user must have a valid digital identity issued by the organization's MSP to submit transactions.

6. Ledger

- The **ledger** is the main record-keeping system in Fabric.
- It consists of two parts — the **blockchain** (immutable sequence of blocks) and the **world state** (a database with current key-value pairs).
- The world state can be stored using **LevelDB** or **CouchDB**.
- Example: In a trade network, the ledger stores transaction details like product ID, quantity, and price.

7. Clients

- Clients are applications or users that interact with the Fabric network.
- They submit transaction proposals to endorsing peers and receive responses.
- Example: A web application for invoice processing acts as a client in a Fabric network.

Transaction Flow in Fabric v1 (Execute–Order–Validate Model)

1. **Proposal Phase:**
    a. A client submits a transaction proposal to the endorsing peers.
    b. The endorsing peers simulate the transaction without updating the ledger and generate **endorsements** (digital signatures).
2. **Ordering Phase:**
    a. The endorsed transactions are sent to the **ordering service**.
    b. The orderer collects and arranges these transactions into **blocks** and delivers them to all peers.
3. **Validation Phase:**
    a. Each peer validates the block by checking endorsement policies and version numbers.
    b. If valid, the transaction is committed to the ledger, and the world state is updated.

Example Workflow

- Suppose a logistics company uses Fabric for package tracking.
- A client sends a proposal: "Update package status to 'Delivered.'"
- Endorsing peers simulate the change and sign it.
- The ordering service collects all proposals and makes a new block.
- All peers validate and commit the block, updating the ledger with the new delivery status.