Week1:

1). Designing the Application Architecture

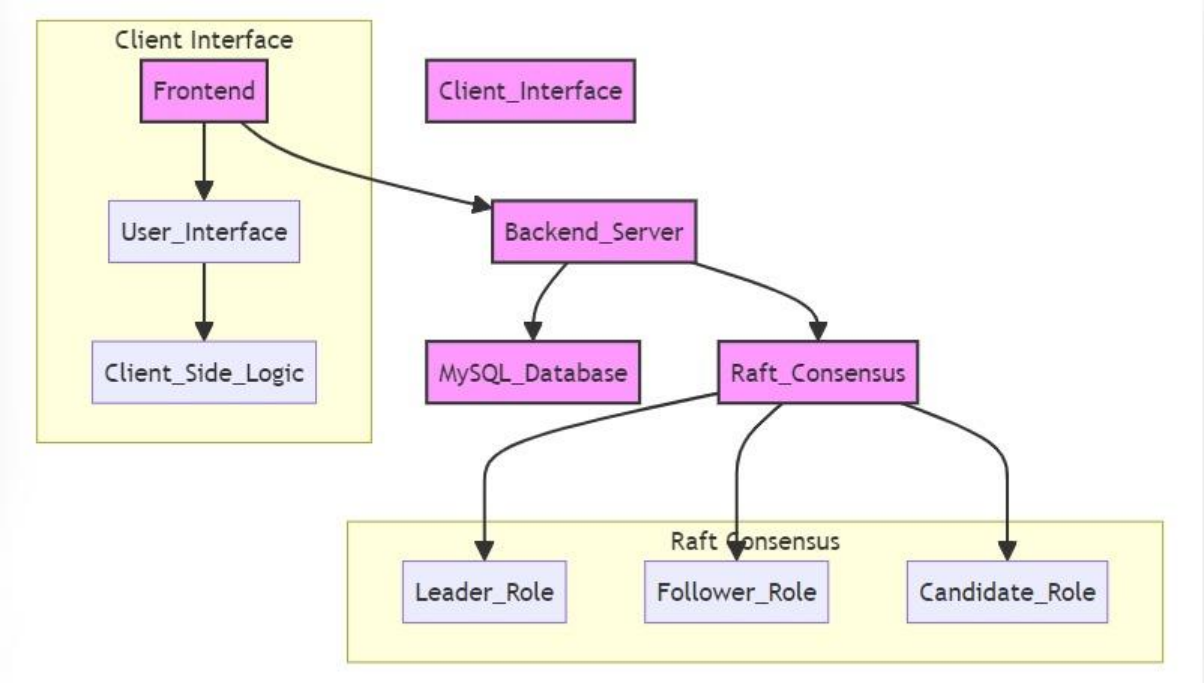
● Define the architecture of the task management application.

User Interface (UI): The UI layer serves as the interface between users and the application. It provides various screens, forms, and controls allowing users to interact with the task management system. This can be implemented using web technologies, mobile frameworks, or desktop applications based on the platform requirements.

Backend: The backend comprises several components responsible for business logic, data storage, and communication with clients.

- **Task Management Logic:** This component encapsulates the core business logic of the application, handling task creation, updates, and deletion, as well as assignment, prioritization, and notifications. It enforces the application's rules and processes related to task management.
- **User Authentication and Authorization:** This component ensures that only authorized users access and interact with the application. It employs mechanisms such as username/password authentication or integration with third-party providers like OAuth. Authorization controls user actions and data access based on predefined roles and permissions.
- **Collaboration and Communication:** This aspect facilitates collaboration among users through features like comments, file attachments, task sharing, and real-time notifications. Integration with messaging platforms or email notifications enhances communication efficiency.
- **Integration and APIs:** The application may offer integration capabilities with external systems and services, such as calendars, project management tools, or communication platforms. APIs enable developers to extend or integrate the application with other services seamlessly.
- **Reporting and Analytics:** This component enables the generation of reports and analytics based on task data. It provides insights like task completion rates, user productivity metrics, and task aging analysis. Reporting tools or data visualization libraries aid in presenting information effectively.

Database: The database component manages the storage and retrieval of task data. It stores essential information such as task details, user profiles, authentication data, comments, and attachments. MySQL is chosen as the backend database due to its relational model, robust querying capabilities, and transaction support.



• Determine the role of each node in the Raft consensus algorithm, such as leader, follower, and candidate.

Raft Node Roles:

In the Raft consensus algorithm, each node plays a specific role in ensuring fault tolerance and maintaining consistency across the distributed system:

- **Leader:** Responsible for managing log replication and coordinating actions across the cluster. It receives client requests, appends them to its log, replicates them to followers, and ensures consensus before committing changes to the database.
- **Follower:** Passive nodes that replicate log entries from the leader and respond to client requests. They participate in leader elections and follow the leader's instructions to maintain consistency.
- **Candidate:** Nodes aspiring to become a leader. Candidates initiate leader elections when they haven't heard from the current leader within a specified time frame

• Plan the schema for storing task data in MySQL, considering factors like table structure, relationships, and indexing.

MySQL Schema Design:

The schema for storing task data in MySQL includes tables and relationships optimized for efficient querying and data manipulation:

- **Tasks Table:** Stores task information such as task ID, name, description, due date, status, assigned user ID, creation timestamp, and last update timestamp.
- **Users Table:** Contains user information including user ID and username.
- **Task Status Table:** Defines various task statuses (e.g., "To Do," "In Progress," "Completed").
- **Task Priority Table:** Defines different task priorities (e.g., "Low," "Medium," "High").

Relationships:

- **One-to-Many Relationship**: Each user can be associated with multiple tasks.
- **Many-to-One Relationship**: Each task can have a single status and priority.
- **Leader Table**: Contains the user ID of the leader node.

Indexes are added on frequently queried columns (e.g., task ID, user ID) to enhance query performance, and foreign key constraints maintain data integrity and enforce relationships between tables.

2). Implementing Raft Consensus Algorithm

● Implement Raft consensus algorithm to ensure fault tolerance and consistency across multiple nodes.

Raft Consensus Algorithm Implementation:

1. **Components:**

    - **Servers:** Application will consist of multiple servers running the Raft algorithm.
    - **Roles:** Each server can be in one of three roles: Leader, Follower, or Candidate.

        - **Leader:** Responsible for replicating data entries to followers and committing them once a majority agrees.
        - **Follower:** Passively receives data entries from the leader and applies them to their local storage.
        - **Candidate:** Servers become candidates if the leader fails and hold elections to elect a new leader.

● Develop communication protocols between nodes for achieving consensus.

Communication Protocols (Remote Procedure Calls):

- **RequestVote RPC:** Used by candidates to gather votes from other servers during elections.
- **AppendEntries RPC:** Used by the leader to replicate data entries to followers and for heartbeat.

● Develop communication protocols between nodes for achieving consensus.

Testing and Validation:

- **Simulate Different Scenarios:** Such as leader failure, network partitions, and follower crashes.
- **Verify Consistency:** Ensure the system maintains consistency (all servers agree on the task list) despite failures.

Task Management Application:

1. **Data Model:**
    - Design a data model in MySQL/NoSQL to store tasks (e.g., task ID, description, status, etc.).
2. **Client Interaction:**
    - Develop a client application (web or mobile) for users to interact with the task management system.
3. **Server-side Logic:**
    - Implement server-side logic to handle task CRUD (Create, Read, Update, Delete) operations.
4. **Raft Integration:**
    - Integrate the Raft implementation with your task management logic.
        - Leader election and communication protocols should ensure all servers eventually agree on the latest task list.
        - Leader replicates task updates (create, update, delete) to followers using AppendEntries RPC.
        - Followers apply these updates to their local storage.

Pseudocode:

server.py 1

D: > Harshitha > CC_proj > server.py > ...

```python
from enum import Enum

class Role(Enum):
    FOLLOWER = 1
    CANDIDATE = 2
    LEADER = 3

class Server:
    def __init__(self, server_id):
        self.server_id = server_id
        self.role = Role.FOLLOWER
        self.current_term = 0
        self.voted_for = None
        self.log = []  # Log entries
        self.communication = RaftCommunication('localhost', 5000 + server_id)  # Assuming each server listens on a different port

    def handle_append_entries_rpc(self, message):
        # Logic to handle AppendEntries RPC (heartbeat or log replication)
        pass

    def handle_request_vote_rpc(self, message):
        # Logic to handle RequestVote RPC (election process)
        pass

    def start_election(self):
        # Logic to start an election
        pass

    def heartbeat(self):
        # Send heartbeat messages to followers periodically
        pass

```

raft_communication.py

D: > Harshitha > CC_proj > raft_communication.py > ...

```python
import socket
import pickle

class RaftCommunication:
    def __init__(self, host, port):
        self.host = host
        self.port = port

    def send_message(self, message, target_host, target_port):
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((target_host, target_port))
            s.sendall(pickle.dumps(message))

    def receive_message(self):
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.bind((self.host, self.port))
            s.listen()
            conn, addr = s.accept()
            with conn:
                data = conn.recv(1024)
                if data:
                    return pickle.loads(data)
```

task_mgt.py 1

D: > Harshitha > CC_proj > task_mgt.py > ...

```python
class Task:
    def __init__(self, id, description, status):
        self.id = id
        self.description = description
        self.status = status

class TaskManager:
    def __init__(self):
        self.tasks = []
        self.server = Server()  # Initialize Raft server

    def create_task(self, task):
        # Logic to create task and send it for Raft consensus
        self.server.update_state(task)

    def get_tasks(self):
        # Logic to retrieve tasks from local database after Raft consensus
        return self.tasks

```