Hochschule Fulda
University of Applied Sciences

Distributed Applications – AI5109

Research and Design

Module - 2

Submitted by: Usama Tahir (1453517)

Project Description

Project Title: Easy Chat

Technology Used

Android Studio: The primary integrated development environment (IDE) for Android app development.

Java/Kotlin: The main programming languages for Android app development.

Firebase: For real-time database, authentication, and cloud messaging services.

Description

The Easy Chat Application is a real-time messaging platform designed for seamless communication between users. The application leverages Android Studio as the development environment and utilizes Firebase services for efficient data storage, authentication, and real-time updates. Users can create accounts, log in securely, and engage in instant messaging with others.

Logic

User Authentication: Users can create accounts or log in using Firebase Authentication, ensuring secure and personalized access to the chat platform.

Real-time Database: Firebase Realtime Database is employed to store chat messages, user information, and other relevant data. This allows for instant updates and synchronization across all connected devices.

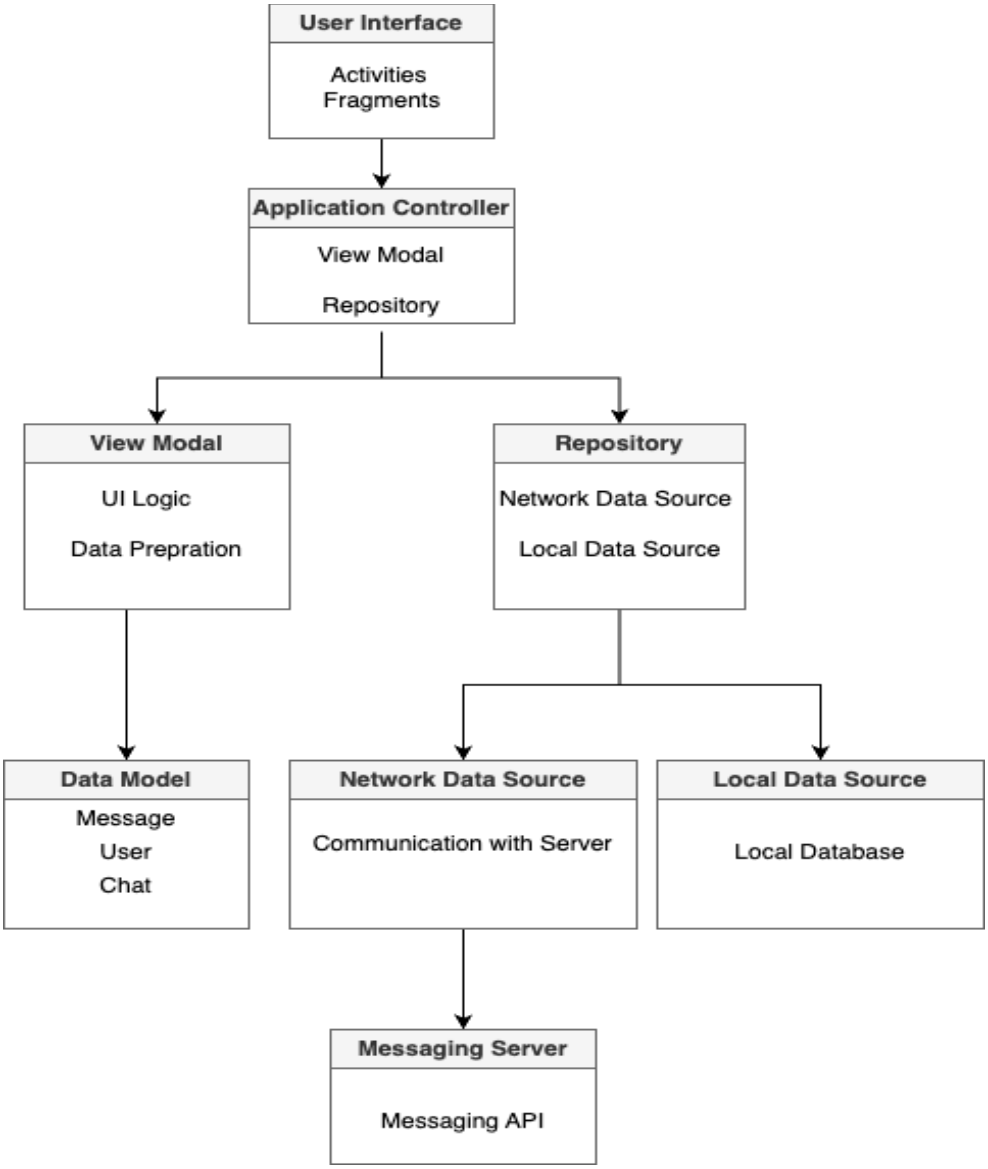
Chat Interface: The application provides an intuitive chat interface where users can send and receive text messages in real time. Multimedia attachments like images or files can be supported.

Push Notifications: Firebase Cloud Messaging is used to implement push notifications, ensuring users are promptly notified of new messages even when the app is in the background.

UML Diagrams

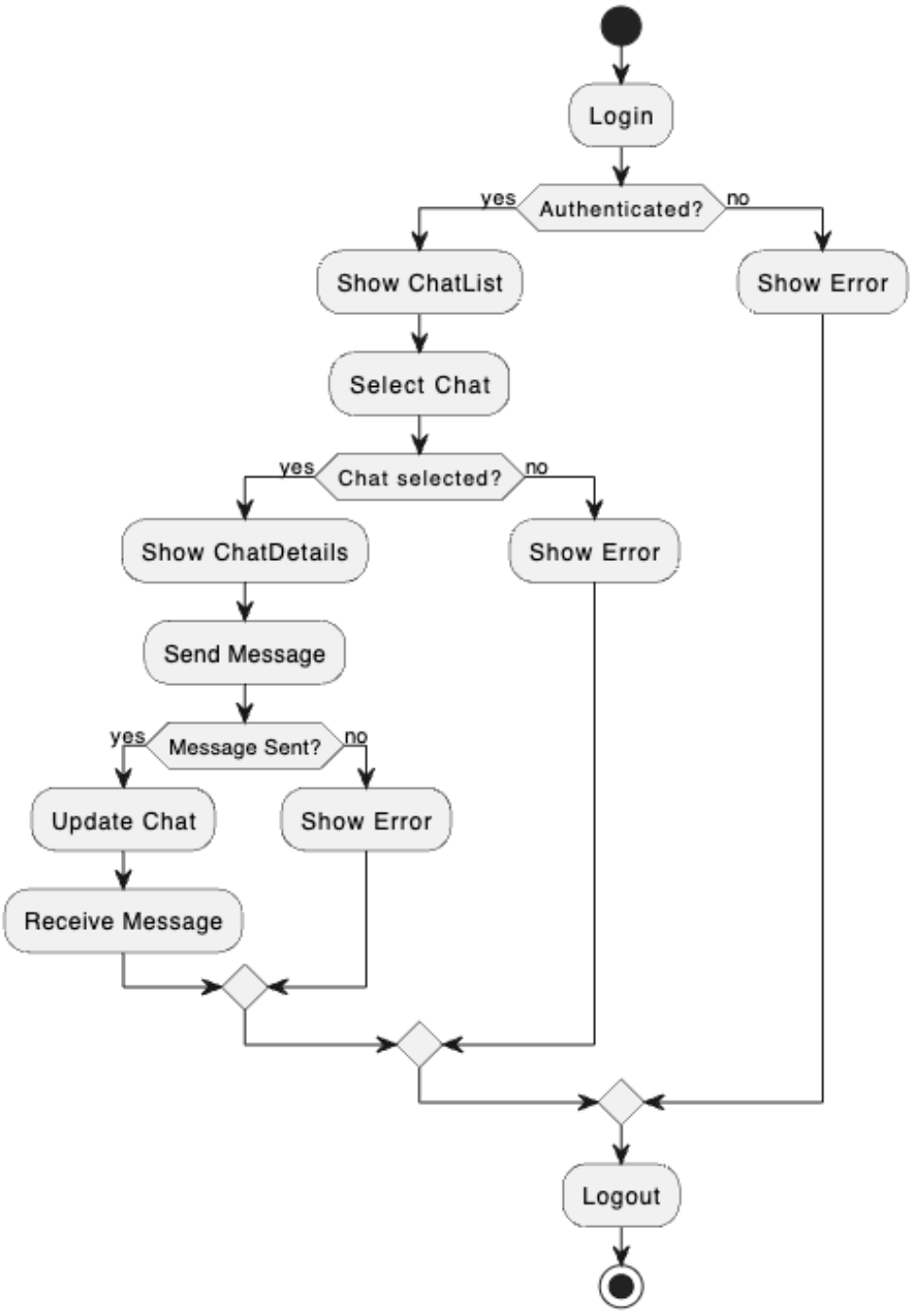
Architecture Diagram

The architecture diagram illustrates the organization and interactions among key components in the Android chat application, delineating the relationships between the user interface, backend services, local database, and the cloud-based messaging server



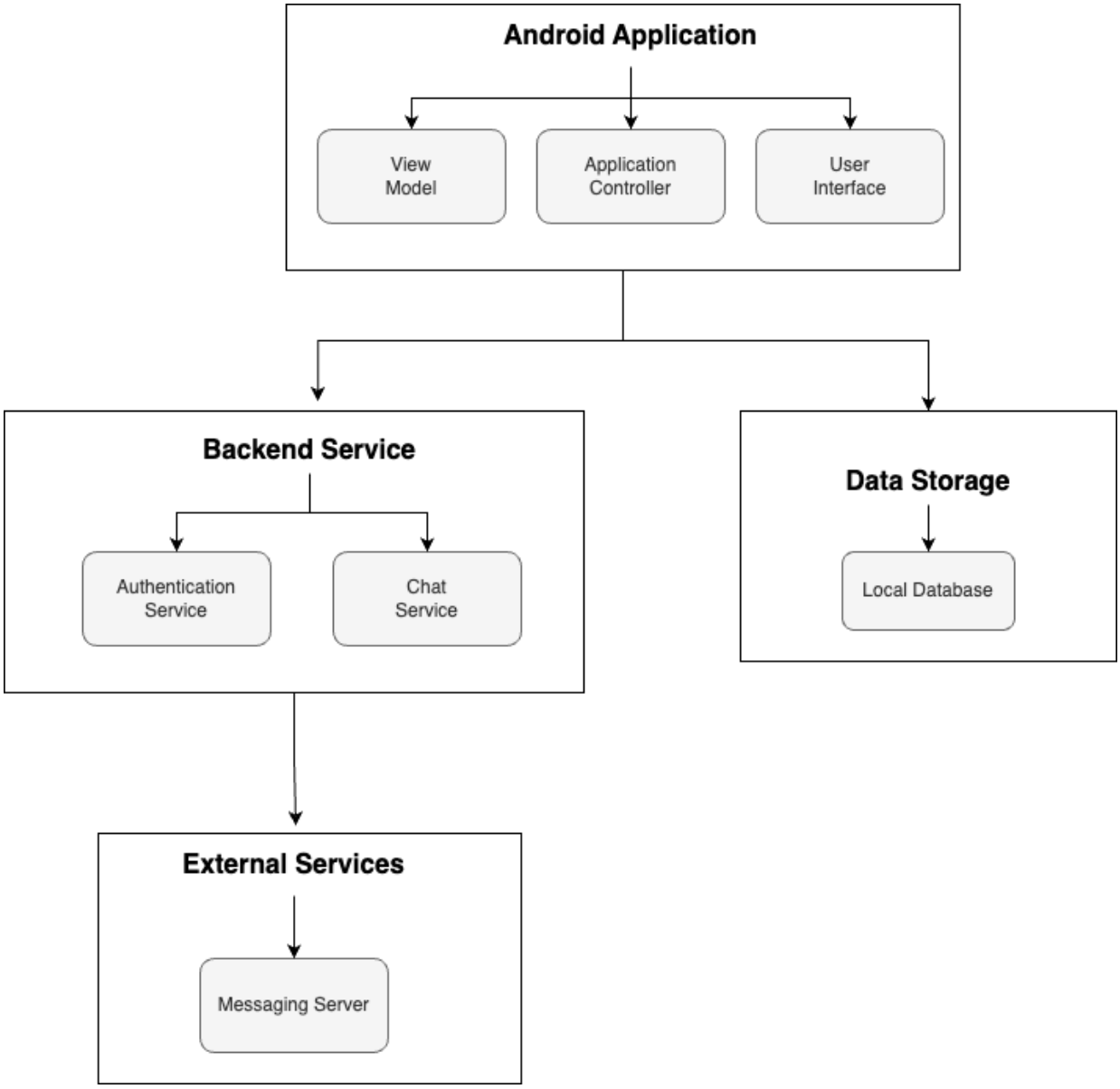
Activity Diagram

Activity diagram visually depicts the dynamic flow of user interactions within the chat application, outlining key activities such as login, chat selection, message sending, and receiving. It provides a sequential representation of user actions, offering insights into the application's functionality and user experience



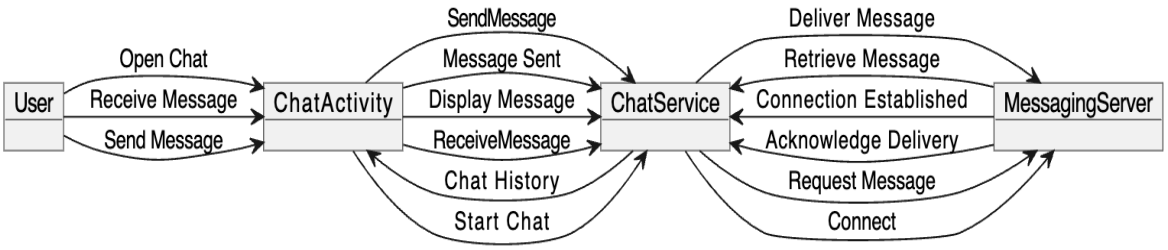
Component Diagram

Component diagram illustrates the modular structure of the chat application, showcasing the key components like user interface, backend services, local database, and messaging server, clarifying their relationships and dependencies. It provides a high-level overview of the application's architecture, aiding in system understanding and design communication



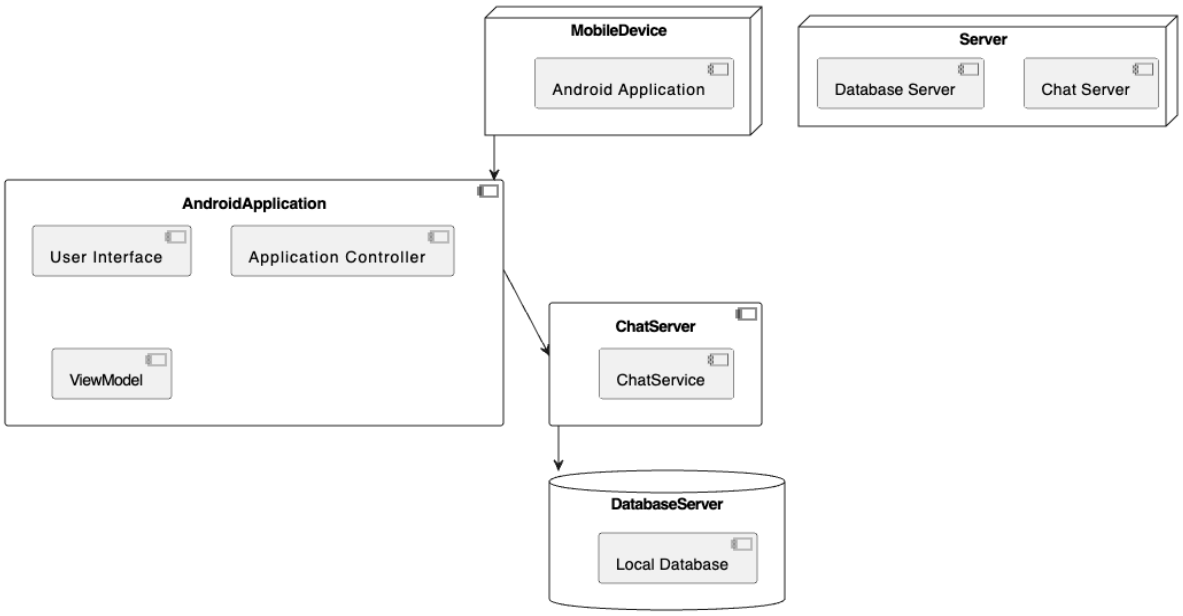
Communication Diagram

Communication diagram visually represents the dynamic exchange of messages and interactions between components in the Android chat application, emphasizing the seamless flow of communication among the user, chat activity, service components, and the messaging server



Deployment Diagram

The deployment diagram illustrates the physical distribution and arrangement of software components, showcasing the deployment of the Android chat application on mobile devices and servers. It provides insights into the deployment structure, facilitating understanding of how the application's components are situated across different hardware environments.



Brief answers to the questions

Explain EJB, JMS MDB in detail

Enterprise JavaBeans (EJB)

Enterprise JavaBeans (EJB) is a server-side component architecture for building scalable, distributed, and transactional enterprise-level applications in Java. EJB provides a framework for developing and deploying distributed, transactional, and secure enterprise applications. There are three types of EJBs

Session Beans

- Session beans represent business processes or workflows.
- They are used to implement the business logic of an application.
- There are two types: Stateful and Stateless.

Entity Beans

- Entity beans represent persistent data in a database.
- They are used to model and manage business data.
- With the advent of EJB 3.0, the concept of Entity Beans has been largely replaced by JPA (Java Persistence API) for handling persistence.

Message-Driven Beans (MDB)

- Message-Driven Beans are special EJBs designed for asynchronous processing.
- They consume messages from a JMS (Java Message Service) queue or topic.
- They are often used for implementing background processing, event-driven systems, and decoupling components in a distributed system.

Java Message Service (JMS)

Java Message Service (JMS) is an API that provides a common way for Java programs to create, send, receive, and read messages asynchronously between loosely coupled, distributed systems. JMS supports both point-to-point (queues) and publish/subscribe (topics) messaging models.

Key Concepts in JMS:

- Message: The unit of data sent between JMS clients.
- Queue: A point-to-point destination where messages are sent and consumed by a single receiver.
- Topic: A publish/subscribe destination where messages are sent and consumed by multiple subscribers.

Message-Driven Beans (MDB)

Message-Driven Beans (MDB) are a type of EJB specifically designed to work with JMS. They allow Java EE applications to process messages asynchronously, making them suitable for handling background tasks, event-driven architectures, and integrating disparate systems.

Key Characteristics of MDB:

- Annotation-Based: In modern EJB versions, MDBs are often annotated to simplify configuration.
- JMS Integration: MDBs are associated with a JMS destination (queue or topic) and automatically receive and process messages.
- Asynchronous Processing: MDBs allow for the asynchronous handling of messages, promoting decoupling in distributed systems.

Explain Replication

Replication, in the context of databases and distributed systems, refers to the process of copying and maintaining the same data across multiple servers or nodes. The primary goal of replication is to improve data availability, fault tolerance, and performance. There are various replication models and strategies, but they all involve creating and maintaining copies of data in multiple locations. Here are some key concepts related to replication

Data Replication Models

Master-Slave Replication

- In a master-slave replication model, there is one primary server (master) that handles both read and write operations.
- The changes made on the master are asynchronously or synchronously propagated to one or more slave servers.
- Read operations can be distributed among the slaves, improving overall read performance.

Multi-Master Replication

- In a multi-master replication model, multiple servers can accept both read and write operations.
- Changes made on one master are propagated to other masters, ensuring that all masters have the same data.
- Multi-master replication can enhance write scalability and fault tolerance.

Peer-to-Peer Replication

- Peer-to-peer replication allows all nodes to accept both read and write operations.
- Changes are distributed among all nodes in a decentralized manner.
- This model can offer high scalability and fault tolerance.

Synchronous vs. Asynchronous Replication

Synchronous Replication

- In synchronous replication, a write operation is not considered complete until it has been replicated to all designated nodes.
- This ensures that all replicas are consistent at the cost of increased latency for write operations.

Asynchronous Replication

- In asynchronous replication, a write operation is considered complete as soon as it is committed on the primary node, without waiting for replication to other nodes.
- Asynchronous replication can provide lower write latency but may introduce temporary data inconsistency between replicas.

Use Cases and Benefits

- High Availability: Replication enhances system availability by providing redundancy. If one node fails, others can continue serving requests.
- Load Balancing: Replicas can distribute read traffic, improving overall system performance and balancing the load on individual nodes.
- Disaster Recovery: Replication supports data recovery in the event of hardware failures, data corruption, or disasters.

Challenges

- Consistency: Ensuring consistency among replicas, especially in the presence of concurrent writes, can be challenging.
- Latency: Synchronous replication can introduce latency for write operations due to the need for confirmation from multiple nodes.

Explain Distributed File Systems

Distributed File Systems (DFS) are designed to provide a unified and coherent view of a file system across multiple, geographically dispersed nodes in a network. The primary goal is to enable efficient and reliable file access and storage in a distributed environment. Distributed File Systems offer features such as scalability, fault tolerance, and data consistency. Here are some key concepts and characteristics

Distributed Namespace

In a distributed file system, the namespace (the hierarchy of directories and files) is distributed across multiple servers or nodes.

Each node in the network has a part of the namespace, and together they form a unified, global namespace.

File Replication

To enhance fault tolerance and availability, distributed file systems often replicate files across multiple nodes.

Replication helps in maintaining multiple copies of a file, and if one node fails, clients can still access the data from replicas.

Consistency and Coherency

Ensuring consistency and coherence of data across distributed nodes is a significant challenge. Distributed file systems implement protocols and mechanisms to maintain data consistency, even in the presence of concurrent access and updates.

Scalability

Distributed file systems are designed to scale horizontally to accommodate growing amounts of data and increasing numbers of clients.
Scalability is achieved by adding more nodes to the system.

Fault Tolerance

DFS provides fault tolerance by replicating data and by distributing it across multiple nodes.
If a node fails, clients can access replicas or data from other nodes.

Access Transparency

Distributed file systems aim to provide access transparency, meaning that users and applications can access files without being aware of the underlying distribution of data across nodes.

Security

Security in the context of distributed systems is a critical and complex aspect that involves protecting the confidentiality, integrity, and availability of data and services across multiple interconnected nodes. Distributed systems face unique challenges due to factors such as the heterogeneity of the network, the potential for malicious attacks, and the need to ensure secure communication and coordination among nodes. Here are key aspects of security in distributed systems

Authentication

Challenge: Verifying the identity of nodes in a distributed system.
Solution: Use authentication mechanisms like digital certificates, tokens, or credentials to ensure that nodes can be trusted.

Authorization

Challenge: Controlling access to resources based on the identity and permissions of the entities.
Solution: Implement access control mechanisms to specify who is allowed to perform what actions on which resources.

Data Encryption

Challenge: Ensuring the confidentiality of data transmitted over the network.
Solution: Use encryption algorithms and protocols to secure data in transit and at rest. This includes securing communication channels and encrypting stored data.

Integrity

Challenge: Ensuring that data is not tampered with during transmission or storage.
Solution: Implement integrity checks, such as checksums or digital signatures, to detect and prevent unauthorized modifications to data.

Auditing and Logging

Challenge: Monitoring and tracking activities to detect and respond to security incidents.
Solution: Implement auditing and logging mechanisms to record relevant events, providing visibility into system activities and potential security breaches.

Distributed Denial of Service (DDoS) Protection

Challenge: Mitigating the impact of DDoS attacks that attempt to overwhelm a system by flooding it with traffic.
Solution: Employ DDoS protection mechanisms, such as traffic filtering, rate limiting, and distributed traffic analysis, to prevent or minimize disruptions.

Secure Communication

Challenge: Ensuring secure communication between distributed nodes.
Solution: Use secure communication protocols, such as TLS/SSL, to encrypt data during transmission and to authenticate communicating parties.

Secure Transactions

Challenge: Ensuring the security of distributed transactions, especially in scenarios involving multiple nodes and data updates.
Solution: Implement distributed transaction protocols and two-phase commit mechanisms to ensure consistency and atomicity.

Firewalls and Intrusion Detection Systems (IDS)

Challenge: Protecting the distributed system from external and internal threats.
Solution: Deploy firewalls to control network traffic and intrusion detection systems to identify and respond to potential security breaches.

Security Patching and Updates

Challenge: Ensuring that all nodes in the distributed system are running updated and patched software.

Solution: Establish a robust system for applying security patches and updates across the distributed nodes to address vulnerabilities.

Consensus Algorithms

Challenge: Achieving agreement among distributed nodes in the presence of faulty or malicious nodes.
Solution: Implement consensus algorithms, such as Paxos or Raft, to ensure agreement and consistency in distributed decision-making.

Fault Tolerance and Redundancy

Challenge: Maintaining system availability despite failures or attacks.

Solution: Design the distributed system with fault tolerance and redundancy, ensuring that critical functions can continue even if some nodes are compromised.

Container

In the context of distributed systems, a container is a lightweight and portable unit that encapsulates an application and its dependencies. Containers provide a consistent and isolated environment for running applications across various computing environments. They are designed to be easily deployable, scalable, and maintainable, making them well-suited for distributed architectures. Here are key characteristics and components of containers in distributed systems

Key Characteristics

Isolation
Containers encapsulate an application and its dependencies, providing isolation from the host system and other containers.
Each container runs in its own user space, ensuring that the application's environment is consistent and independent.
Portability
Containers are portable and can run consistently across different environments, including development, testing, and production.
The encapsulation of dependencies ensures that the containerized application behaves predictably regardless of the underlying infrastructure.
Efficiency
Containers share the host operating system's kernel, making them lightweight and resource-efficient compared to traditional virtualization.
Multiple containers can run on the same host without the overhead of running separate operating system instances.
Packaging
Containers package an application and its dependencies into a single, standardized unit known as a container image.

The container image includes the application code, runtime, libraries, and other configurations needed for the application to run.

Scalability

Containers support easy scaling of applications by replicating containers horizontally. Container orchestration tools, such as Kubernetes, can manage the deployment and scaling of containerized applications across a distributed cluster.

Explain what is Docker and Kubernetes

Docker

Docker is an open-source platform that enables developers to automate the deployment of applications inside lightweight, portable containers. Containers encapsulate an application and its dependencies, ensuring consistency across different environments and simplifying the deployment process.

Containerization

Docker uses containerization technology to package applications and their dependencies into a single unit called a Docker container.
Containers are isolated from the host system and each other, providing consistency and portability.

Docker Images

A Docker image is a lightweight, stand-alone, and executable package that includes the application code, runtime, libraries, and other settings required for the application to run.
Images are used to create Docker containers.

Docker Hub

Docker Hub is a cloud-based registry that allows developers to share and distribute Docker images.
Developers can pull pre-built images from Docker Hub or push their custom images for others to use.

Docker file

Docker file is a script used to create a Docker image.
It contains instructions for building the image, specifying the base image, adding dependencies, and configuring the application.

Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications.
It uses a YAML file to configure the application's services, networks, and volumes.

Use Cases

Application Packaging

Docker simplifies the packaging and distribution of applications, ensuring they run consistently across different environments.

Microservices
Docker is often used in microservices architectures, where each microservice is encapsulated in a container.

Kubernetes:

Kubernetes (often abbreviated as K8s) is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications. It provides a robust and extensible framework for managing containerized workloads in a distributed environment.

Container Orchestration
Kubernetes automates the deployment, scaling, and management of containerized applications.
It handles tasks such as container placement, health monitoring, scaling, and rolling updates.
Nodes and Clusters
Kubernetes clusters consist of a master node and multiple worker nodes (minions).
Worker nodes run containers, and the master node manages the overall cluster.
Pods
A pod is the smallest and simplest unit in the Kubernetes object model.
It represents a single instance of a running process in a cluster and can contain one or more containers.
Services
Kubernetes Services provide a stable endpoint (IP and port) to access a set of pods.
They enable load balancing and abstract the network connectivity between pods.
Deployments
Deployments in Kubernetes define the desired state for the application, including the number of replicas and the container image.
Kubernetes ensures that the actual state matches the desired state.
Ingress
Kubernetes Ingress is an API object that manages external access to services within a cluster.
It provides HTTP and HTTPS routing to services based on rules.

Use Cases

Container Orchestration

Kubernetes is widely used for automating the deployment, scaling, and management of containerized applications.
Microservices Architecture
Kubernetes is a popular choice for managing microservices-based applications.
Cloud-Native Development
It is a key technology in the cloud-native ecosystem, supporting modern development practices.