HLD Case study 3 (Messaging)

Contents

[System Design Process Overview 2](#_Toc184643806)

[Gathering Requirements/MVP for Messaging App 4](#_Toc184643807)

[Estimation of Scale 6](#_Toc184643808)

[API Design 9](#_Toc184643809)

[Message System Design 12](#_Toc184643810)

[Sharding and Database Design for Messaging Systems 14](#_Toc184643811)

[Message Consistency Across UserID based Shards 19](#_Toc184643812)

[Message System Caching 22](#_Toc184643813)

## System Design Process Overview

This lecture outlines a four-step approach to system design, covering requirement gathering, scale estimation, API design, and detailed design.

**Step 1: Gathering Requirements / MVP**

* **Objective:** Understand the system's functional and non-functional requirements.
  + Functional requirements: What the system must do (features, behaviors).
  + Non-functional requirements: Constraints like performance, scalability, reliability, etc.
* **MVP Focus:** Identify the minimum viable product (MVP) to build an initial version that satisfies critical requirements.

**Step 2: Estimating Scale**

* **Key Questions to Address:**
  + **Sharding:** Do we need to distribute data across multiple databases or servers?
  + **System Load:** Is the system predominantly read-heavy or write-heavy?
    - Read-heavy systems prioritize fast data retrieval.
    - Write-heavy systems optimize for data insertion or updates.
* **Capacity Estimation:**
  + Project data volume, query load, and user traffic.
  + Decide on infrastructure needs (e.g., horizontal/vertical scaling).

**Step 3: API Design**

* **Purpose:** Define clear interfaces for communication between system components or with external clients.
* **Best Practices:**
  + Focus on usability, consistency, and versioning.
  + Ensure APIs align with both functional and scale considerations.

**Step 4: Detailed Design**

* **Objective:** Dive into the low-level implementation details of the system.
* **Key Considerations:**
  + Choose appropriate architecture patterns (e.g., microservices, monolith).
  + Identify technologies, frameworks, and data models.
  + Address concurrency, fault tolerance, and failover mechanisms.

## Gathering Requirements/MVP for Messaging App

**Part 1: Defining the MVP (Minimum Viable Product)**

**Functional Requirements for MVP:**

1. **One-on-One Chat:**
   * Core functionality: Send and receive messages between two users.
   * Include a **list of conversations**, displayed in **most recent first** order.
   * When clicking on a conversation, show the **list of messages** for that conversation, also in **most recent first** order.
2. **Conversation List Features:**
   * Display **snippets of the last message** in each conversation.
   * Show **number of unread messages** for each conversation.
   * Include **timestamps** for messages (e.g., time sent or received).
3. **Message Screen Features:**
   * When entering a conversation, display:
     + **Profile picture** of the user.
     + Chat history (to-and-fro messages).
   * Enable sending new messages within the conversation.
4. **Notifications:**
   * Notify users when a new message is received.
5. **Basic User Management:**
   * **User Registration:** To create accounts and enable communication.
   * **Contact List:** To identify recipients for messages.

**Features Excluded from MVP:**

* **Read Receipts (e.g., blue ticks).**
* **Media Sharing:** Images, audio, emojis.
* **Calls:** Voice or video calling.
* **Group Chats:** Prioritize one-on-one chat initially.
* **Message Reactions or Likes.**
* Advanced options like:
  + Blocking/unblocking users.
  + Message deletion.
  + Money transfer within the app.

**Non-Functional Requirements**

**Primary Design Goals:**

1. **Low Latency:**
   * Messages should be sent and received in near real-time for a seamless user experience.
2. **High Consistency vs. High Availability:**
   * **Consistency:**
     + Ensure messages are delivered in order.
     + Avoid cases where:
       - A newer message arrives before an older one.
       - Only partial messages (e.g., the latest message but not prior ones) are visible.
     + Rationale: Disordered or missing messages could lead to misunderstandings or conflicts.
     + Decision: Prioritize **high consistency** over availability for this system.
   * **Availability:** While important, availability takes a back seat to ensure message accuracy and proper sequencing.
3. **Scalability:**
   * Prepare the system to handle growing user bases and traffic while maintaining performance.

**Additional Notes:**

* **Consistency Trade-offs:**
  + Facebook's approach: Emphasized high consistency to reduce risks of miscommunication.
  + Other messaging apps like WhatsApp may prioritize high availability, especially in multi-device scenarios.
* **WhatsApp Example:**
  + Earlier versions emphasized high consistency for single-device setups.
  + Current multi-device implementations may involve different trade-offs.
* **Derived Requirements:**
  + Characteristics like **read-heavy vs. write-heavy** load and infrastructure scaling needs emerge from the functional and non-functional requirements and overall system design.

## Estimation of Scale

**1. Goals of Scale Estimation**

* **Sharding**: Determine if the system needs to distribute data across multiple storage units.
* **Read/Write Characteristics**: Evaluate whether the system is:
  + Read-heavy (more data retrieval than storage).
  + Write-heavy (more data storage than retrieval).
* **Capacity Planning**: Estimate the number of machines needed to handle the workload.

**2. Steps to Estimate Scale**

* **Total Storage Calculation**:
  1. Determine **what is being stored** (e.g., messages).
  2. Estimate **daily usage**:
     + Number of messages sent per day.
     + Average size of each message.
  3. Calculate storage needs:
     + Daily storage = daily message count × average message size.
     + Annual storage = daily storage × 365.
     + Long-term storage (e.g., 10 years) = annual storage × years.
* **Assumptions**:
  1. For Facebook Messenger:
     + Approx. 20 billion messages/day.
     + Average message size ~200 bytes. (Discussed later why 200 bytes)
* **Storage Estimation Example**:
  1. Daily storage .
  2. Annual storage .
  3. Storage for 10 years .

**3. Sharding**

* Required when storage needs exceed the capacity of a single machine or database instance.
* In the example, 15 PB of storage necessitates sharding.

**4. Read vs. Write Characteristics**

* **Scenario**: Messaging between two users:
  + Sending a message = 1 write.
  + Reading the message = 1 read.
* **Balance**:
  + For every message sent, there is typically one read.
  + Volume of reads and writes is nearly equal.
  + Reads may slightly outnumber writes due to additional retrieval operations (e.g., viewing older messages).

**5. Challenges in Balancing Reads and Writes**

* Systems that are both read- and write-heavy require careful optimization.
* Solutions to handle high loads:
  + **Batching Writes**: Temporarily store writes and sync them periodically (used in less-consistent systems).
  + **Optimizing Reads**: Reduce the number of read operations hitting the database by using caching or indexing.

**6. Message Metadata**

* Components of a stored message:
  + **Sender ID**: 8 bytes.
  + **Recipient ID**: 8 bytes.
  + **Timestamp**: 8 bytes.
  + **Message ID**: 8 bytes.
  + **Message Text**: ~100 bytes (average).
  + **Metadata**: ~60 bytes (e.g., read status, edits, additional properties).
* **Total Message Size**:
  + Approx. .

**7. Key Insights**

* Sharding is essential for large-scale systems.
* Messaging systems are typically **both read- and write-heavy**.
* Efficient system design requires:
  + Careful database selection.
  + Strategies to limit either reads or writes based on system requirements.
* Consistency and availability trade-offs influence design choices.

**Capacity Planning**

1. **Importance of Capacity Planning**:
   * Estimate machine requirements based on QPS (Queries Per Second) and workload per query.
   * Identify bottlenecks: CPU, memory, network.
   * Decide the number of machines required to process the expected workload.
2. **Differentiating Systems by Workload**:
   * **Read-heavy**: Content like Instagram posts (few writes, many reads).
   * **Write-heavy**: Systems like logging services (frequent writes, rare reads).
   * **Balanced (Read/Write-heavy)**: Systems where reads and writes are comparable.

## API Design

**1. Send Message API**

**Purpose:**

* Allows a client to send a message to a recipient while ensuring reliability and avoiding duplicates.

**Key Points:**

1. **Parameters**:
   * **Sender ID**: Identifies the user sending the message.
   * **Recipient ID**: Identifies the user or group receiving the message.
   * **Message Text**: The content of the message.
   * **Message ID**: A unique identifier for each message.
2. **Timestamp Generation**:
   * **Generated on the server**:
     + Avoids trust issues with client-side timestamps (e.g., clock errors, malicious tampering).
     + Ensures consistency and prevents spoofing.
   * **Stored as an Epoch Timestamp**:
     + Epoch timestamps (seconds since January 1, 1970, UTC) ensure uniformity across time zones.
3. **Importance of Message ID**:
   * Ensures **idempotency**:
     + Retries due to network failures won’t create duplicate messages.
   * Uniqueness achieved using:
     + Timestamp.
     + User ID.
     + Device ID.
   * **UUID/Guid Libraries**:
     + Typically used to generate unique IDs on the client-side.

**2. Get Conversations API**

**Purpose:**

* Fetches the list of conversations for a user, including the most recent messages.

**Key Points:**

1. **Parameters**:
   * **User ID**: Identifies the user for whom conversations are fetched.
   * **Limit**: The number of conversations to retrieve (e.g., 10 or 15).
   * **Offset**: Used for pagination (starting point for fetching conversations).
   * **Timestamp (for delta fetch)**:
     + Used in **mobile apps** for incremental updates since the last fetch, reducing bandwidth usage.
2. **Delta Fetch**:
   * Ensures only updated conversations are fetched after the provided timestamp.
   * Suitable for clients with local storage, such as mobile apps.
3. **Two Approaches**:
   * **Full Fetch**: Fetches all recent conversations (e.g., desktop web apps without local storage).
   * **Delta Fetch**: Fetches only the updates after a specific timestamp (e.g., mobile apps).

**3. Get Messages API**

**Purpose:**

1. Fetches the messages within a specific conversation.

**Key Points:**

1. **Parameters**:
   * **Conversation ID**: Identifies the specific conversation whose messages need to be fetched.
   * **Limit**: The number of messages to fetch (e.g., for pagination).
   * **Offset**: Specifies the starting point for fetching messages.
2. **Pagination**:
   * Similar to the Get Conversations API, supports fetching messages in batches.

**General Design Principles:**

1. **Idempotency**:
   * Critical for APIs like sendMessage to avoid duplicates during retries.
2. **Efficiency**:
   * Delta fetch reduces bandwidth usage for mobile apps.
   * Local storage is leveraged to minimize server calls.
3. **Consistency**:
   * Server-generated timestamps ensure uniformity and trustworthiness.
4. **Scalability**:
   * APIs are designed with pagination (limit/offset) for handling large datasets.

**Summary of the Three APIs:**

| **API Name** | **Parameters** | **Use Case** |
| --- | --- | --- |
| sendMessage | Sender ID, Recipient ID, Text, Message ID | Sends a message while ensuring reliability and idempotency. |
| getConversations | User ID, Limit, Offset, Timestamp | Fetches the list of recent conversations (or changes since a given time). |
| getMessages | Conversation ID, Limit, Offset | Retrieves messages for a specific conversation with pagination. |

## Message System Design

1. **Basic Architecture Overview**

* **Clients**:
  + Can be a browser or an app.
  + Used for sending messages or fetching conversations/messages.
* **Load Balancer**:
  + Distributes requests to app servers.
* **App Servers**:
  + Processes client requests (e.g., send messages, fetch conversations/messages).
  + A computer network diagram with a few blue objects

    Description automatically generated with medium confidenceCommunicates with a database to fetch/store data.

1. **Sharding**

**A. Sharding by User ID**

* Each user's data is stored on a specific shard based on their User ID.

**Impact on Key Use Cases**

1. **Send a Message**:
   * Efficient, as messages are written to a single shard (the sender’s).
   * Potential issue: If the recipient’s shard is needed, it adds cross-shard communication.
2. **Get Conversations**:
   * Efficient, since all conversations involving a user are on the same shard.
3. **Get Messages (within a conversation)**:
   * Can be inefficient for group conversations involving multiple users, as data might need to be fetched from multiple shards.

**Advantages:**

* Works well for one-on-one messaging systems.
* User-based queries (e.g., get user’s conversations) are efficient.

**Disadvantages:**

* Cross-shard communication for group conversations.
* Potential hotspot issues if some users are significantly more active than others.

**B. Sharding by Conversation ID**

* Each conversation's data is stored on a specific shard based on the Conversation ID.

**Impact on Key Use Cases**

1. **Send a Message**:
   * Efficient, as all messages in a conversation are stored on the same shard.
2. **Get Conversations**:
   * Requires fetching data from multiple shards to aggregate all conversations for a user.
3. **Get Messages (within a conversation)**:
   * Highly efficient, as all messages for a conversation are on the same shard.

**Advantages:**

* Optimized for group conversations, as all conversation-related data is collocated.
* Easier to scale for large group chats.

**Disadvantages:**

* User-centric queries (e.g., get all conversations for a user) require querying multiple shards.
* Slightly less optimal for one-on-one messaging use cases.

1. **4. SQL vs NoSQL for Sharded Systems**

* **SQL**:
  + Not ideal for heavy sharding due to schema and relational constraints.
  + May face performance bottlenecks.
* **NoSQL**:
  + Better suited for sharded use cases.
  + Flexible schema and efficient for distributed storage.

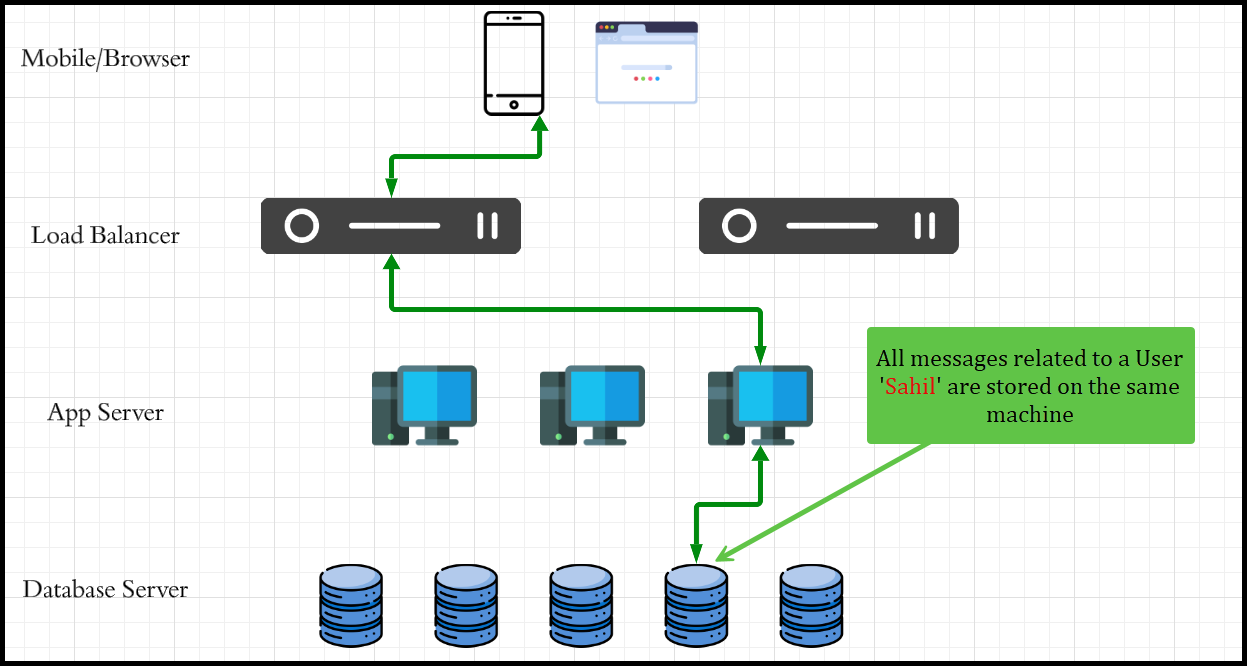
## Sharding and Database Design for Messaging Systems

**1. Introduction to Sharding**

* Sharding splits data across multiple machines to handle large datasets and traffic efficiently.
* Primary use case: **One-on-one messaging**.
* Critical decision: Choosing the **sharding key**.
  + Common options:
    - **User ID**
    - **Conversation ID**

**2. Sharding by User ID**

* **Definition**: All data related to a user (messages, conversations, etc.) is stored on a single machine.
  + Example: If Sahil has conversations with Raji and Saurabh, all their messages are stored on the same machine.
* **Assumption**: Each user's entire mailbox fits on one machine.



**Advantages:**

* **Efficient reads**:
  + **Get Conversations**: Query one machine for all conversations.
  + **Get Messages**: Query one machine for all messages in a specific conversation.
* **Simple access patterns**:
  + Consistent fetch from a single machine.

**Challenges:**

* **Send Message Operation**:
  + Write to multiple machines:
    1. Add the message to the sender's mailbox.
    2. Add the message to the recipient's mailbox.
  + **Consistency risk**: Ensuring both copies of the message are updated simultaneously.

**Summary:**

* **Get Conversation** and **Get Messages** are simple and efficient.
* **Send Message** requires writing to two machines, creating potential consistency challenges.

**3. Sharding by Conversation ID**

* **Definition**: Each conversation (e.g., Sahil-Raji, Sahil-Saurabh) is assigned to a separate machine.
  + Example: Sahil-Raji conversation is on Machine A; Sahil-Saurabh conversation is on Machine B.

**Advantages:**

* **Efficient writes**:
  + **Send Message**: Write only to the machine hosting the conversation.
  + No need to duplicate messages across multiple machines.

**Challenges:**

* **Get Conversations**:
  + Requires querying all machines to fetch all conversations for a user.
  + Slow and inefficient.

A diagram of a computer network

Description automatically generated

**Optimization:**

* Introduce a secondary database (**DB2**) to store:
  + A recent list of conversations for each user.
  + Timestamps and snippets for each conversation.

**Implications:**

* Sending a message now involves:
  1. Writing the message in the primary database (DB1).
  2. Updating the timestamp and snippet in DB2 for both the sender and the recipient.

A diagram of a computer network

Description automatically generated

* **Write amplification**: One message leads to three writes across different machines.

**Summary:**

* **Send Message** is efficient.
* **Get Conversations** is slow without optimization but becomes manageable with DB2.
* High complexity due to multiple writes and risk of inconsistency.

**4. Comparison of User ID vs. Conversation ID as Sharding Keys**

* **User ID**:
  + Simpler architecture.
  + Lower number of writes (2 writes for Send Message).
  + Better for one-on-one communication.
* **Conversation ID**:
  + Optimized for large group conversations.
  + Higher write complexity (3 writes for Send Message).
  + Suitable for group-based apps like **Slack** or **Discord**.

**Conclusion**: For one-on-one communication, **sharding by User ID** is the better choice due to fewer writes and simpler consistency management.

**5. Choosing a Database**

* Requirements:
  + High write throughput.
  + High consistency.
  + Support for recent conversations and efficient queries.

**Options:**

1. **Key-Value Stores**:
   * Example: Redis.
   * Not suitable for complex queries.
2. **Column Family Stores**:
   * Example: **Cassandra**, **HBase**.
   * Best for write-heavy and consistent systems.
   * Allows efficient queries for most recent conversations/messages.
3. **Document Stores**:
   * Example: MongoDB.
   * Better for flexibility but not as optimized for high write volumes.

**Preferred Solution:**

* **Column Family Database (HBase)**:
  + **Write-heavy optimization**:
    - Appends write operations to a Write-Ahead Log (WAL) instead of applying them immediately.
  + Slow reads but highly consistent writes.
  + Works well when reads are cached.

**6. Key Points to Remember**

* Sharding key choice depends on the primary use case:
  + One-on-one messaging: **User ID**.
  + Large group messaging: **Conversation ID**.
* **HBase** is a strong candidate for handling high write volumes with consistency.
* Balancing **write amplification** and **read performance** is critical for scalability.

## Message Consistency Across UserID based Shards

**Scenario:**

* A message is sent from **Sahil** to **Rajiv**.
* Two separate database writes are required:
  1. Write to **Sahil's DB (outbox)**.
  2. Write to **Rajiv's DB (inbox)**.

A diagram of a computer network

Description automatically generated

**Key Challenges:**

1. **Write Failure and Inconsistency:**
   * One write might succeed while the other fails.
   * This results in inconsistent states:
     + Message visible in **Sahil's** DB but not in **Rajiv's** DB, or vice versa.
   * User experience issues:
     + Message appears sent but is not delivered.
     + Messages may appear out of order.
2. **Retry Logic:**
   * Retrying a failed write can lead to further complications:
     + Other messages might get written in the meantime, disrupting ordering.
     + Delayed retries might still result in out-of-order messages.
3. **Immediate vs Eventual Consistency:**
   * Eventual consistency (e.g., retry mechanism) may not guarantee the right user experience.
   * Immediate consistency demands a design that minimizes visible inconsistencies.

**Proposed Solution: Write Order**

**Two Approaches:**

**1. Write to Sahil's DB First:**

* **Case 1: Success on Sahil's DB, failure on Rajiv's DB:**
  + Sahil sees the message in his DB (local success).
  + Rajiv doesn't receive the message due to failure.
  + Sahil refreshes and still sees the message, creating the illusion of success.
  + If rollback is implemented:
    - The message is removed from Sahil's DB, causing further confusion.
* **Case 2: Failure on Sahil's DB:**
  + Easy to notify Sahil of the failure.
  + No inconsistency.

**Drawback:** Inconsistent user experience in the success-failure scenario:

* Sahil believes the message is sent when it isn't.

**2. Write to Rajiv's DB First:**

* **Case 1: Success on Rajiv's DB, failure on Sahil's DB:**
  + Rajiv receives the message (notification sent).
  + Sahil's DB doesn’t reflect the message due to failure.
  + Sahil knows he sent the message, but it disappears from his view upon refresh.
  + **Recovery:**
    - Retry the write to Sahil's DB.
    - Temporary inconsistency, but the message is successfully delivered to the recipient.
* **Case 2: Failure on Rajiv's DB:**
  + Notify Sahil immediately of the failure.
  + No inconsistencies.

**Advantage:**

* Better user experience:
  + Sahil is aware of failures immediately.
  + Delivered messages appear in correct order for Rajiv.

**Drawback:**

* Temporary inconsistency where Sahil doesn’t see the message in his DB until retry succeeds.

**Conclusion: Prefer Writing to Rajiv’s DB First**

**Reasons:**

1. **Better User Experience:**
   * Inconsistencies are less severe.
   * Rajiv receives messages correctly and in order.
   * Sahil’s experience reflects reality (failure or temporary invisibility).
2. **Recovery Mechanism:**
   * Retry mechanism can resolve temporary inconsistencies for Sahil’s DB.
3. **Lesser Evil:**
   * The inconsistency of **"Sahil sees the message but Rajiv doesn’t"** is worse than **"Rajiv gets the message but Sahil doesn’t see it immediately."**

## Message System Caching

**1. Consistency in Messaging Systems**

* **Consistency Problem:**
  + **Scenario 1:** A message appears in the sender's mailbox but never reaches the recipient (bad consistency).
  + **Scenario 2:** A message reaches the recipient but does not reappear in the sender’s mailbox on refresh (better inconsistency).
  + **Preferred Approach:** Prioritize delivering messages to the recipient before updating the sender’s mailbox.
* **Solution:**
  + Write messages to the recipient first.
  + Show failure if the write to the recipient fails.
  + Update the sender’s mailbox after ensuring delivery to the recipient.

**2. Media Messages**

* Media files are not passed around directly.
* **Process:**
  1. Media files are uploaded to object storage or a CDN.
  2. A URL or path is generated and stored.
  3. The URL is attached to the message object and passed along.

**3. Database and Caching**

**HBase as the DB**

* Handles high write volumes.
* Reduces reads to maintain performance (achieved through caching).

**Caching Strategies**

* **Goal:** Minimize DB reads while maintaining consistency.
* **Preferred Cache:** Write-through cache to ensure cache and DB are always in sync.

**Read Handling via Cache**

* Cache the most recent conversations and messages for active users.
* Example: Cache 20 recent conversations and their messages to avoid frequent DB reads.
* **Cache Layers:**
  + **Local/Distributed Cache:** Store user-specific data in the app server memory.
  + A computer diagram of a server

    Description automatically generated with medium confidence**Global Cache (e.g., Redis):** Limited space; used strategically.

**Handling Failures in Cache**

* If a cache node dies:
  + Use **consistent hashing** to map requests to the next available cache node.
  + Missed data triggers a DB fetch, temporarily slowing performance.
  + Cache is "warmed up" as data is retrieved and stored.

**4. Architecture for Caching**

* **Consistent Hashing:** Assign specific users to specific app servers based on a hash.
* **Distributed Cache Memory:** Combines memory of all app servers to increase caching capacity.
* **Fault Tolerance:** If an app server dies, requests shift to the next server. Initial requests might be slow due to cache misses.

**5. WhatsApp vs. Messenger**

* **Messenger:**
  + Stores all historical messages.
  + Accessible across devices and platforms (e.g., website, mobile app).
  + Requires permanent message storage in the database.
* **WhatsApp:**
  + Primarily designed for mobile.
  + Messages stored locally on the user’s device.
  + Servers act as a queue, temporarily holding messages until delivery.
  + Messages older than 7 days are not retained by WhatsApp servers.

**Notes from Video Lecture Transcript**

**1. Understanding Differences in Architectures: WhatsApp vs. Messenger**

* **WhatsApp:**
  + Stores only the most recent messages.
  + Primary focus is on pushing messages to the end devices quickly.
* **Messenger:**
  + Stores all messages.
  + Requires a harder architecture due to the need for comprehensive message storage.

**2. Caching Mechanism in App Servers**

* **Key Challenges:**
  + Limited capacity for caching all conversations and messages.
  + Need to decide what to cache, how much to cache, and the eviction policy.
* **Proposed Caching Strategy:**
  + **User-Level Caching:**
    - Cache most recent 15 conversations.
    - Introduce levels:
      * Level 1: Most recent 15 conversations.
      * Level 2: Conversations 15–30, 30–45, etc., for scrolling.
  + **Conversation-Level Caching:**
    - Cache recent 15 messages per conversation.
    - Use levels for messages as well:
      * Level 1: Most recent 15 messages.
      * Level 2: Messages 15–30, 30–45, etc.
* **Eviction Policy:**
  + Evict older messages first, starting with Level 2 caches.
  + If further eviction is required:
    - Remove older conversations/messages before removing entire user data.
  + Use **Least Recently Used (LRU)**:
    - Implement LRU at conversation level or user level as needed.

**3. Pros and Cons of Using App Servers for Caching**

* **Advantages:**
  + Horizontally scalable: Adding more app servers increases caching capacity.
* **Disadvantages:**
  + **Stateful Architecture:**
    - Each user is tied to a specific app server.
    - Requests fail during server downtimes until reassignment occurs.
  + **Cold Cache Issues:**
    - If a machine dies, a new machine will start with a cold cache, causing initial delays.
  + **Small Window of Unavailability:**
    - Time is required to detect and handle app server failure.
    - During this time, user requests may fail.
  + **Load Skew:**
    - Low probability but possible if many busy users are assigned to the same server.
    - Mitigation: Add more servers to redistribute load.