# The Session Model: Architecting Conversations at Scale

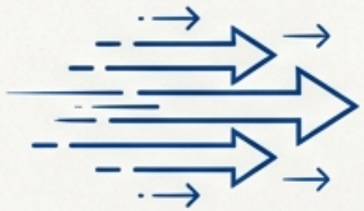
A deep dive into the design principles and architecture of our real-time session management system.

# Why a Message Queue Isn’t Enough

While a message-queue or ‘user mailbox’ model seems simple, it fails to address the functional complexity and performance demands of a modern chat experience.

## The Message Queue Model (The Limitations)

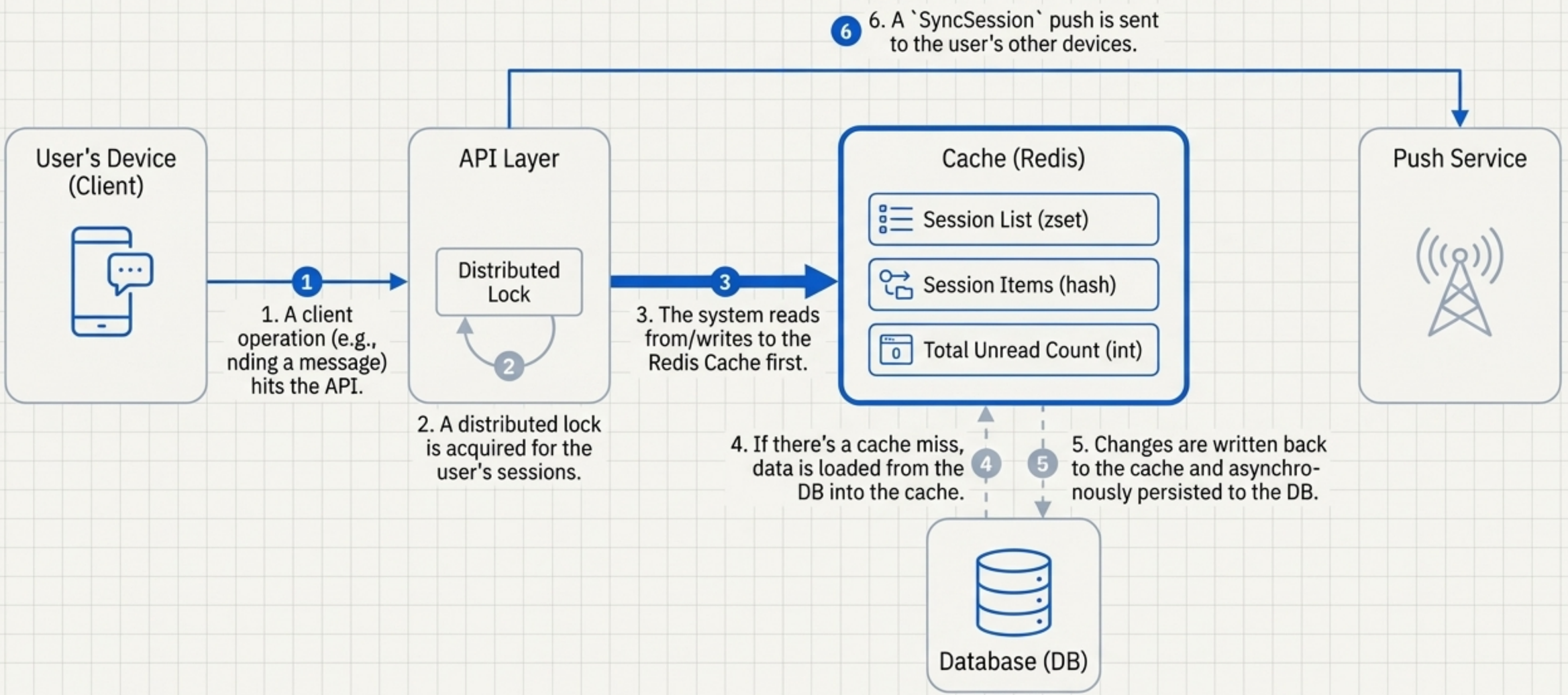
- **Slow User Experience:** On login, the client must pull all incremental messages and reconstruct the session list locally. This is slow, especially with high message volume.
- **Complex Client Logic:** The client becomes responsible for calculating unread counts, sorting, and session state from a raw message stream.
- **State Management is Difficult:** How do you handle session-level actions like 'Delete Chat," ‘Mute," or ‘Mark Unread’? These are not messages. Mixing signals with messages pollutes the model.
- **Inefficient Data Handling:** Deleting a session requires complex cleanup of messages in the server-side mailbox to prevent them from being re-synced.



## The Session Model (The Advantages)

- **Blazing Fast Sync:** Users sync a small, lean list of changed sessions, not a massive queue of messages. This makes the app feel instantly responsive on launch.
- **Rich Functionality is Native:** Session-level attributes (priority, mute status, read state, category) are stored on the server, simplifying client logic and enabling powerful features.
- **Declarative State:** The session is a ‘record’ that gets updated (a replicative, overwrite mechanism), not an immutable log. This aligns with eventual consistency and simplifies state management.
- **Optimized Storage:** The session model is far more storage-efficient than write-diffusing every message to every recipient's mailbox.

# Our Session Architecture: Cache-First by Design



# The Anatomy of a Session

A **Session** is a user-specific record representing a chat's context. For a chat between A and B, A has their own session object and B has theirs. They are independent.

## Session Item

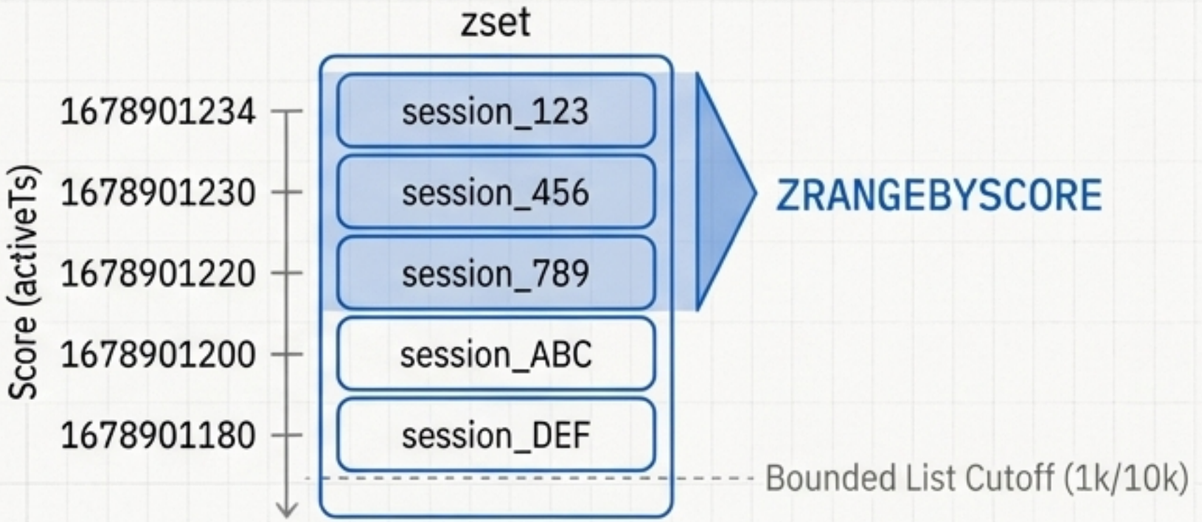
| | |
|---|---|
| `sessionId` | Unique identifier for the conversation. |
| `targetId` | The other user's ID or the group ID. |
| `unreadCount` | Number of unread messages. |
| `readSeq / writeSeq` | The sequence IDs of the last read and last written message. |
| `activeTs` | The absolute timestamp of the last *modification*. **Used for server-side incremental sync.** |
| `writeTs` | The timestamp of the last *user-impacting* modification. **Used for client-side sorting.** |
| `priority` | A flag indicating if the session is pinned/stickied. |
| `categoryId` | Used to group sessions into folders (e.g., "Subscriptions"). |
| `muteStatus` | A boolean for whether the session is muted. |
| `unreadStatus` | A flag for the "Mark as Unread" feature. |
| `version` | A counter to force clients to reload a session's message list. |

# Caching Strategy: The Redis `zset` as a High-Speed Index

## Primary Mechanism

We use a Redis sorted set (`zset`) as the user's "session list" index.

- Key: `session_list:<userId>`
- Member: `sessionId`
- Score: `activeTs` (the timestamp of the last modification)




## The Sync & Load Process

1. **Incremental Sync:** When a client syncs, it provides the last `activeTs` it saw. The server performs a ZRANGEBYSCORE on the `zset` to return only the sessions that have changed.
2. **Bounded List:** The `zset` is capped at the most recent 1k (social) or 10k (workplace) sessions. This is a critical performance assumption: users rarely interact with sessions older than this. This prevents fetching massive, slow lists.
3. **Cache Miss:** If the `sessionList` cache is empty (e.g., an inactive user), we load the most recent 10k sessions from the DB into the `zset`. We explicitly *avoid* a full DB query for users with >500k sessions, as the full-table scan and sort would be prohibitively slow, `update_time`.
4. **Item Fetching:** Individual `sessionItem` data is stored in separate Redis keys. If a requested item isn't in the cache (or was pushed out of the top 10k), it's fetched directly from the DB.

# The Two Timestamps: Sorting for Experience vs. Syncing for Consistency

**Guiding Principle:** The client-side session list order must prioritize the user's sense of importance, which is not always the same as the absolute server-side operation time.

| Action | Updates `writeTs` (Changes Sort Order) | Updates `activeTs` (Triggers Sync) | Rationale |
|---|---|---|---|
| New Message Received | ☑ | ☑ | The most important event; must bring the session to the top. |
| Set Session as 'Unread' | ☑ | ☑ | Explicit user action to re-surface a session. |
| Pin/Unpin Session | ☑ | ☑ | Pinning is a primary organizational tool. |
| Sub-session update in an Aggregate folder | ☑ | ☑ | The folder needs to reflect activity within it. |
| Entering a session / Reading messages | ⊗ | ☑ | Session state changes, but the list shouldn't re-order as you read. |
| Opponent reads your message | ⊗ | ☑ | Your list shouldn't jump around based on others' actions. |
| Muting / Unmuting | ⊗ | ☑ | A background state change; does not warrant a disruptive re-sort. |
| Setting custom `extra` data | ⊗ | ☑ | Purely metadata; invisible to the user. |

The server syncs *all* changes using `activeTs`. The client intelligently re-sorts its list *only* when `writeTs` changes, creating a stable and intuitive user experience.

# Core Operations: The Logic of Read, Delete, and Mute

## Session.Read

- Clears the session's `unreadCount` to zero.
- Adjusts the user's `totalUnreadCount`.
- Updates the session's `readSeq` to the current `writeSeq`.
- Clears any reminder flags (`reminder` data).
- Updates `activeTs` to sync the change, but **not** `writeTs`.
- **Special Case:** Handles 'Mark as Unread' state (`unreadStatus`), allowing a read operation to proceed and reset timestamps even if `readSeq` is already current.

## Session.Remove (Soft Delete)

- Sets the session status to `invalid`.
- Clears `unreadCount` and adjusts `totalUnreadCount`.
- Sets `readSeq` equal to `writeSeq`.
- Updates `activeTs` to sync the deletion.
- **Aggregate Sessions:** If deleting an aggregate folder, all child sessions are recursively soft-deleted first.

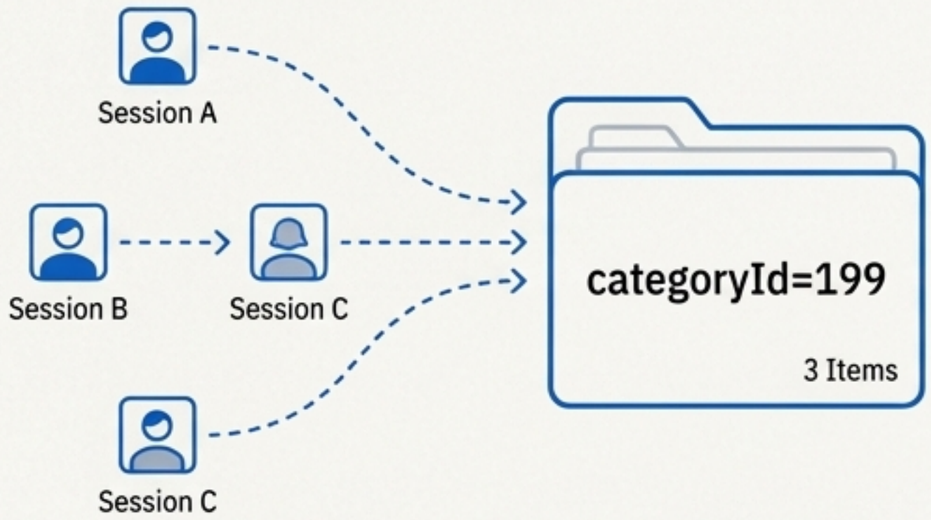
## Session.Mute / Unmute

- Toggles the `muteStatus` flag on the session item.
- **On Mute:** If the session had an unread count, that amount is **subtracted** from the user's `totalUnreadCount`.
- **On Unmute:** The session's current `unreadCount` is **added** back to the `totalUnreadCount`.
- Updates `activeTs` to sync the state change across devices.

# Beyond the List: Managing Complexity with Aggregated Sessions

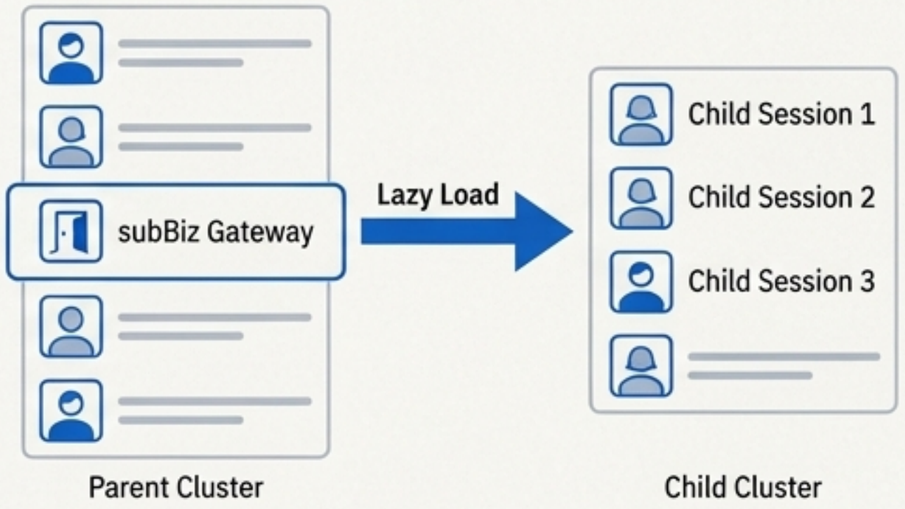
The system supports grouping individual sessions into ‘folders’ or ‘aggregate entries’ to de-clutter the main list and organize conversations. We use two primary mechanisms.

## categoryId Aggregation (Logical Grouping)




- **How it Works:** Sessions are assigned a numerical `categoryId`. The client UI can then 'fold' all sessions with a non-zero `categoryId` (e.g., `categoryId=199` for 'Service Notifications') under a single virtual entry.
- **Server-Side:** A physical 'aggregate session' item may exist to represent the folder. Any update to a child session also triggers an update to the parent aggregate session, causing it to re-sort.
- **Use Cases:** Muting conversations into a 'Message Box,' grouping official accounts.

## subBiz Aggregation (Hierarchical & Physical Grouping)

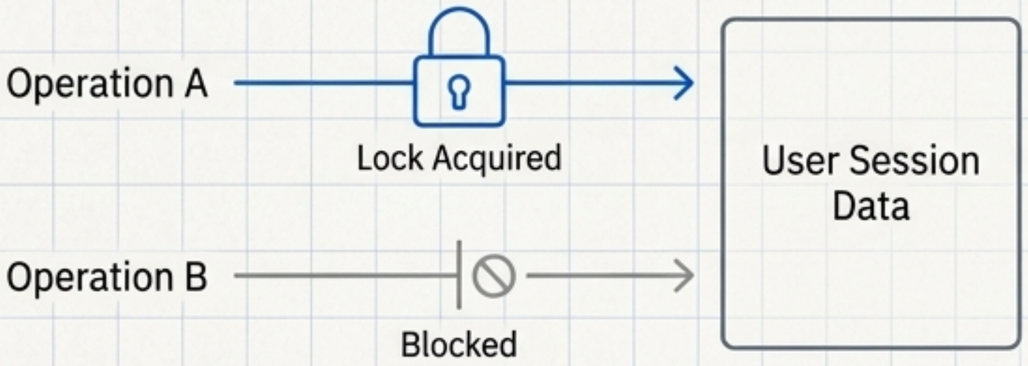



- **How it Works:** A more advanced model for nesting distinct business contexts, like e-commerce chats inside a social media app. A `subBiz` session is a physical entity that acts as a gateway to another, separate session list.
- **Server-Side:** The `targetId` of the aggregate session is a string representing the `subBiz` ID. This session lives in the 'parent' business cluster, while the child sessions exist in a separate 'child' cluster.
- **Performance:** The child list is loaded lazily—only when the user clicks to enter the `subBiz` aggregate session.

# Ensuring Consistency Across a Distributed System

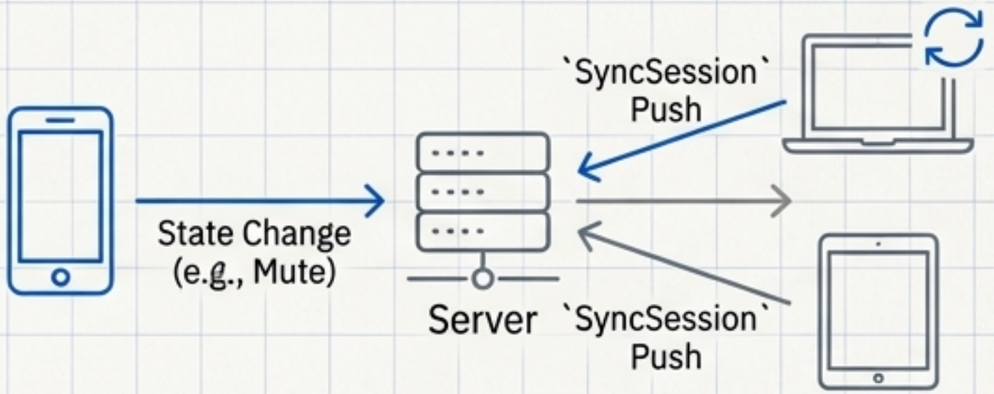
**Challenge:** With multiple operations (e.g., receiving a message, reading a chat) happening concurrently across devices, how do we guarantee data integrity, especially for the `unreadCount`?

## Solution 1: Distributed Locks




- **Scope:** A distributed lock is acquired for a user's session data before any write operation.
- **Purpose:** This ensures that updates to a session item and the corresponding `totalUnreadCount` are atomic. It prevents race conditions where two operations could result in a final incorrect value.
- **Implementation:** A standard distributed lock mechanism (e.g., via Redis or Zookeeper) with a timeout to prevent deadlocks.

## Solution 2: Proactive Multi-Device Sync (`SyncSession`)

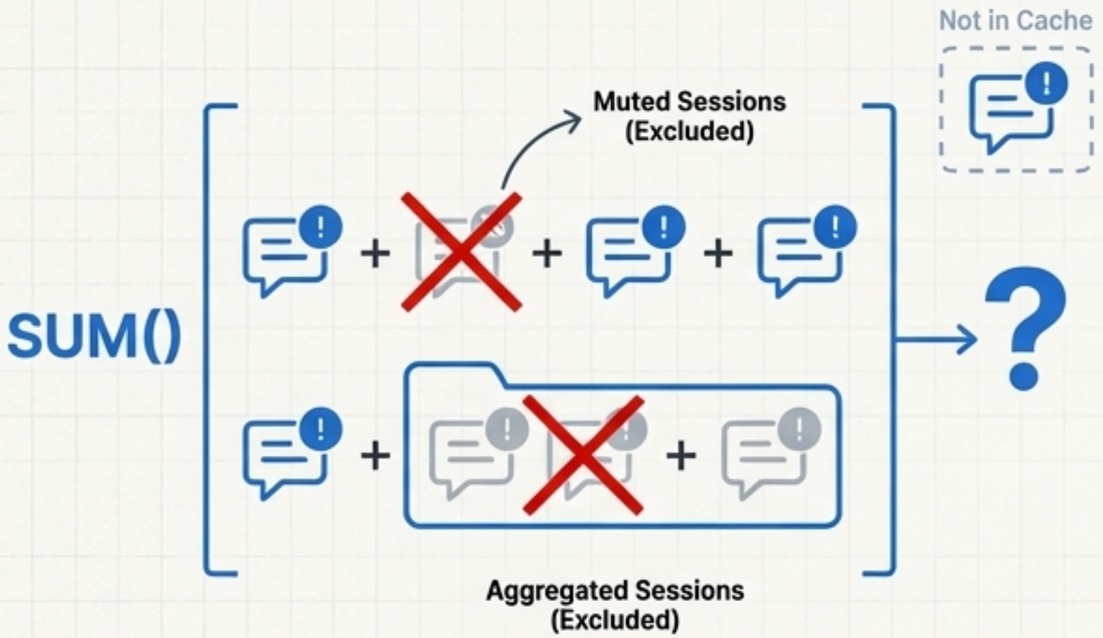



- **Trigger:** Most state-changing operations trigger a lightweight push notification to the user's other active devices.
- **Payload:** The push doesn't contain the full session data, but is simply a signal for the other clients to perform an incremental sync.
- **Operations that trigger `SyncSession`:** Reading messages, Muting or unmuting, Deleting a session, Setting a session to 'Unread', Pinning or unpinning.
- **Note:** New messages have their own push mechanism. `SyncSession` is for state changes *other than* new message arrival.

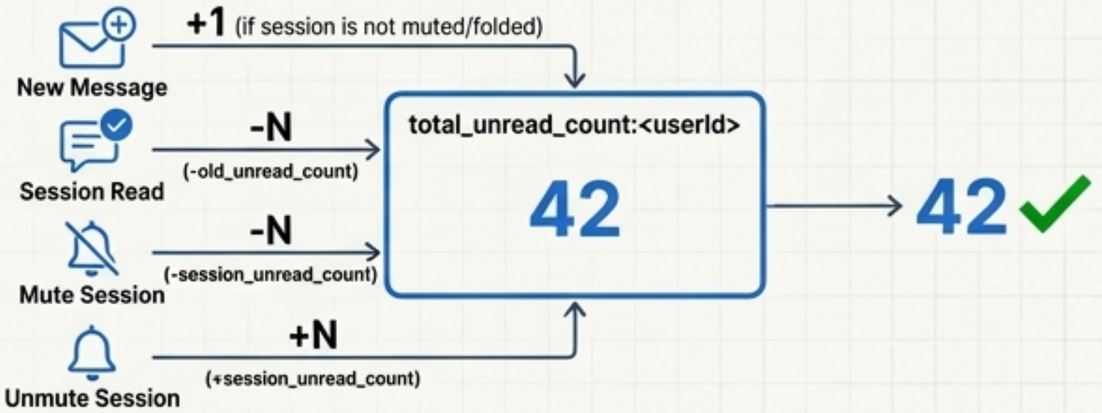
# The Unread Count: More Than a Simple Sum

**The Challenge:** The “total unread count” displayed on the app icon is not simply SUM(unreadCount) for all sessions in the cache.

## Why it's complex:



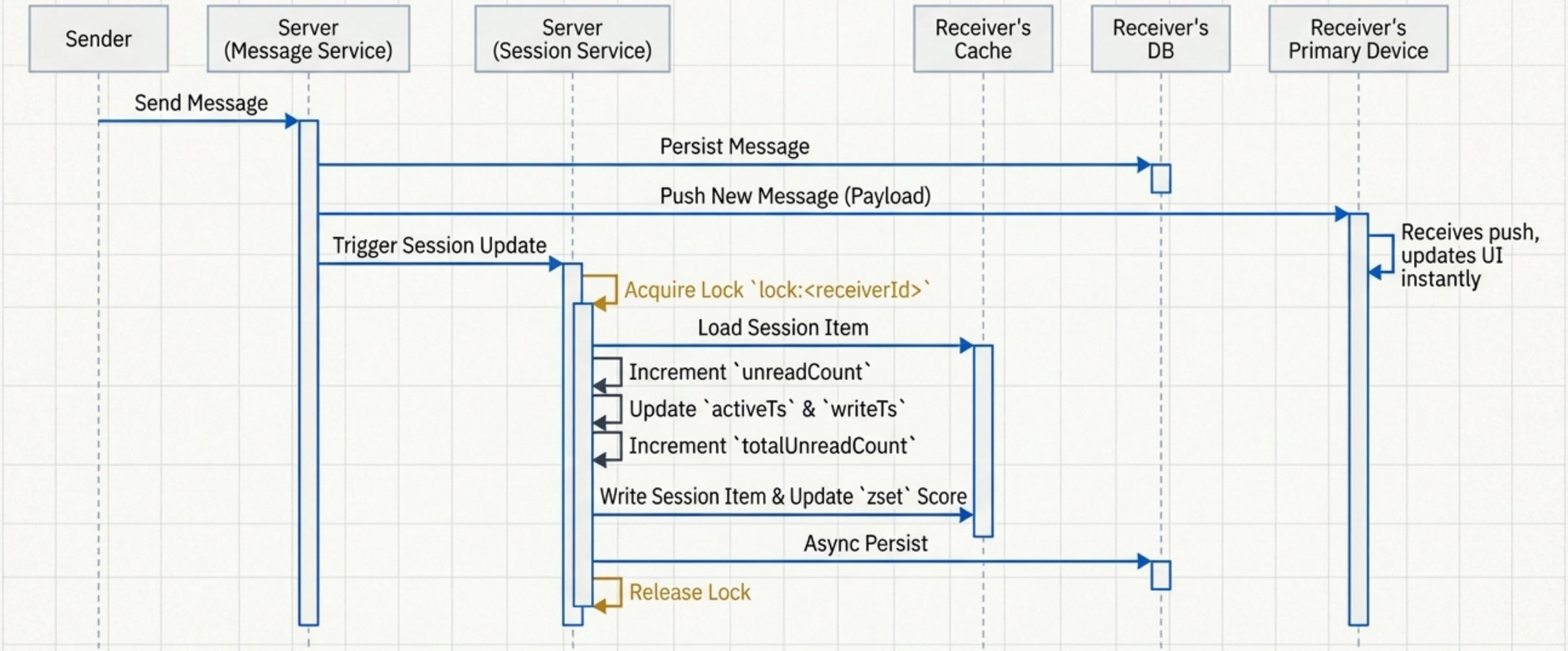

## Our Solution: A Dedicated Counter




- We maintain a separate integer in Redis: **total_unread_count:<userId>**.
- This counter is updated atomically (under a distributed lock) with every relevant operation.
  - **New Message +1** (if session is not muted/folded)
  - **Session Read** - (-old_unread_count)
  - **Mute Session** — (-session_unread_count)
  - **Unmute Session** + (+session_unread_count)
- ****Handling Old Sessions****: When an old session is loaded from the DB into the cache, its unread count is incorporated into the total during the next write operation. This ensures eventual consistency.

**Result:**
The total unread count is always available as a fast, single key lookup, without requiring a costly aggregation query.

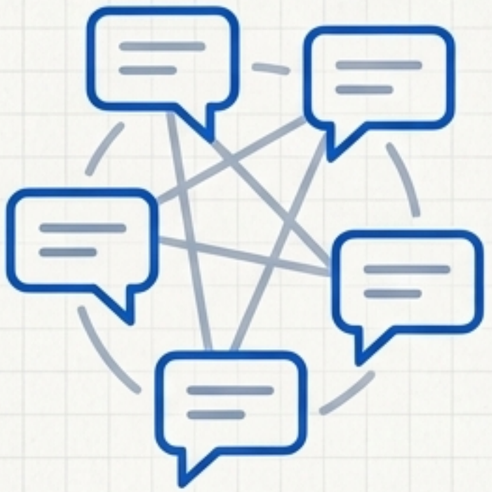
# Architecture in Action: A New Message Arrives




The session update does NOT trigger a separate `SyncSession` push. The new message push is sufficient for other active devices to update their local session state. Inactive devices will get the change on their next incremental sync.

# Engineered to Operate at Internet Scale

## Total Sessions (Private Chat)



**~1 Trillion**
total sessions

**100**
Shards

**10,000**
Tables

**28 TB**
Storage (Single Replica)

## Per-Table Scale (Private Chat)

**11 Million**
records per table (avg)

**2.8 GB**
size per table (2 GB data, 800 MB index)

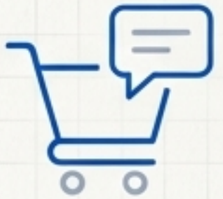
## Group Chat Statistics

**5** members (average)

**76** members (P99)

**500** members (P999)

## Total Sessions (E-commerce Chat)



**~15 Billion**
total sessions

## User Group Count

**2** groups (P50)

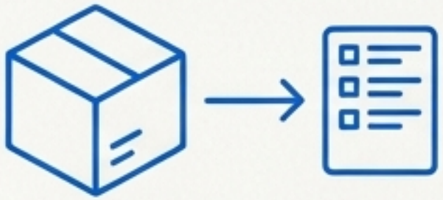
**43** groups (P99)

**7,000** groups (P999)

# Our Core Architectural Decisions & Trade-offs

This system's performance and scalability are the result of conscious design choices that balance user experience, consistency, and operational cost.

## Decision 1: Prioritize User-Perceived Latency.

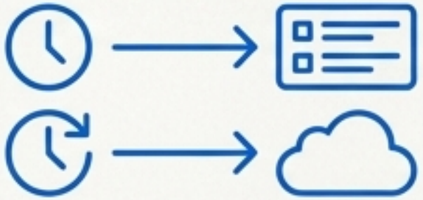


**We Chose:** A cache-first model with a bounded list (top 10k sessions).

**Instead Of:** Querying the full session list from the database on every load.

**The Result:** Near-instant app launch and session list display, at the cost of needing to lazy-load very old sessions on demand.

## Decision 2: Separate UX Sorting from Data Syncing.



**We Chose:** Two distinct timestamps (`**writeTs**` for client sorting, `**activeTs**` for server sync).

**Instead Of:** Using a single timestamp for everything.

**The Result:** A stable, non-jarring UI where the session list only reorders for important events, while ensuring all data changes are still synced reliably across devices.

## Decision 3: A Declarative Session Model over an Event Stream.

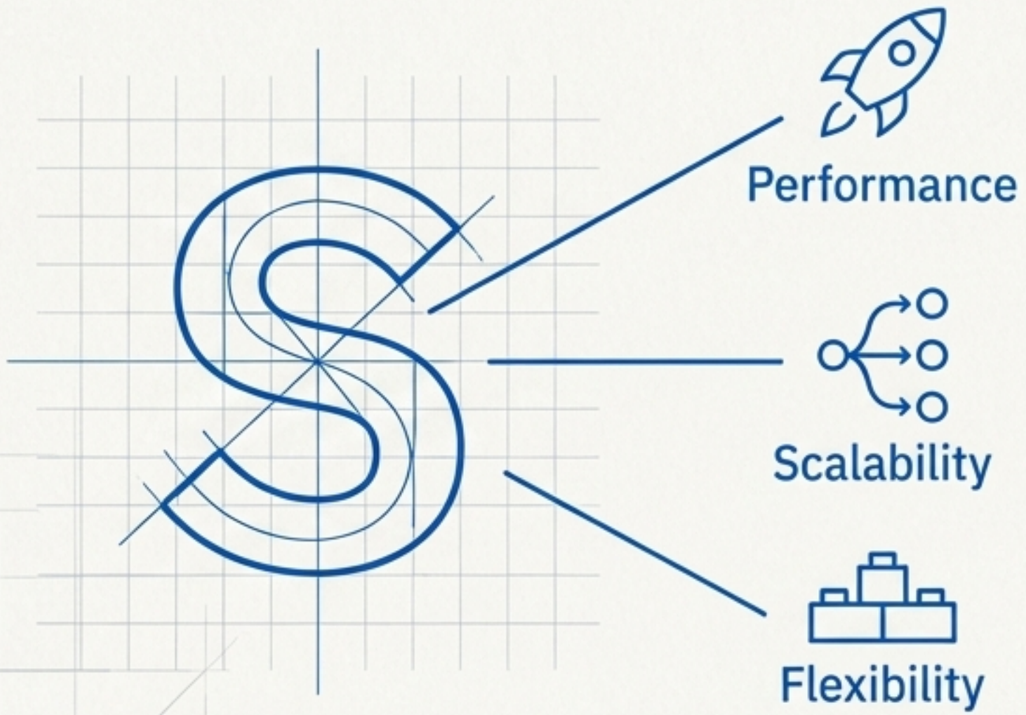
**We Chose:** A model where session state is overwritten (replicative).

**Instead Of:** A pure message queue where clients must derive state.

**The Result:** Dramatically simpler client logic, richer server-side features, and a more efficient storage footprint.

# The Session Model: A Resilient Foundation for Communication

The session management system is more than just a backend component. It is a carefully architected foundation designed for three key outcomes:




1. **Performance**: To deliver an immediate and fluid user experience, even with massive data loads.
2. **Scalability**: To handle trillions of conversations and petabytes of data reliably and cost-effectively.
3. **Flexibility**: To provide a rich data model that supports a growing ecosystem of features, from simple pinning and muting to complex, hierarchical conversation spaces.

By treating the session as a first-class citizen, we have built a system that is not only technically robust but also fundamentally aligned with the user's conversational journey.