The Architecture of a Real-Time Distributed Task Board

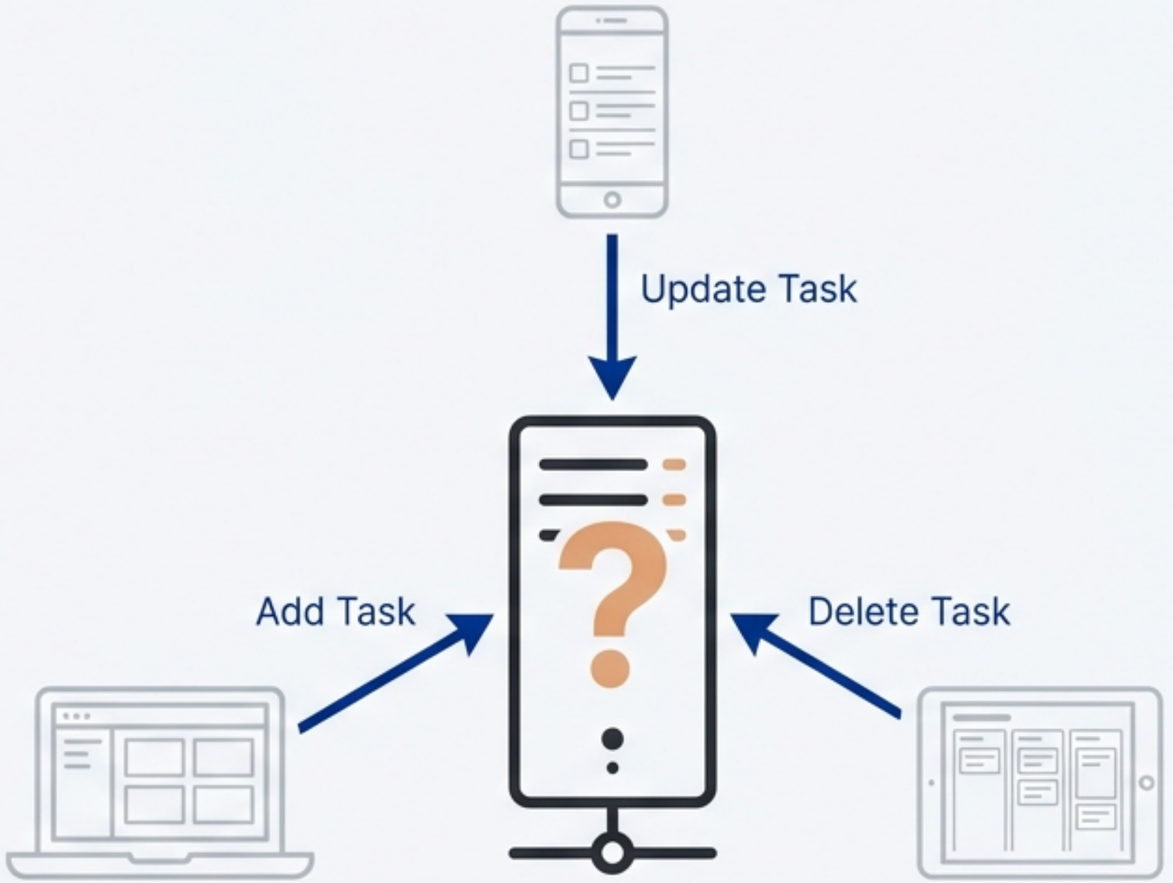
A Deep Dive into Concurrency Patterns and Distributed System Design

The Core Challenge: A Single Source of Truth in a Multi-User World

The Distributed Task Board is a real-time, collaborative application where multiple users can create, update, and manage tasks simultaneously.

The Fundamental Problem:
When multiple users perform actions at the exact same moment, how do we prevent the system from descending into chaos?

- How do we stop data from being corrupted?
- How do we ensure every user sees the exact same state, instantly?
- How do we guarantee operations are consistent and predictable?



Our Blueprint: Building a Robust System from the Ground Up

We will tour the architecture in three logical layers, showing how each layer solves a critical set of problems.

Layer 1: The Connection

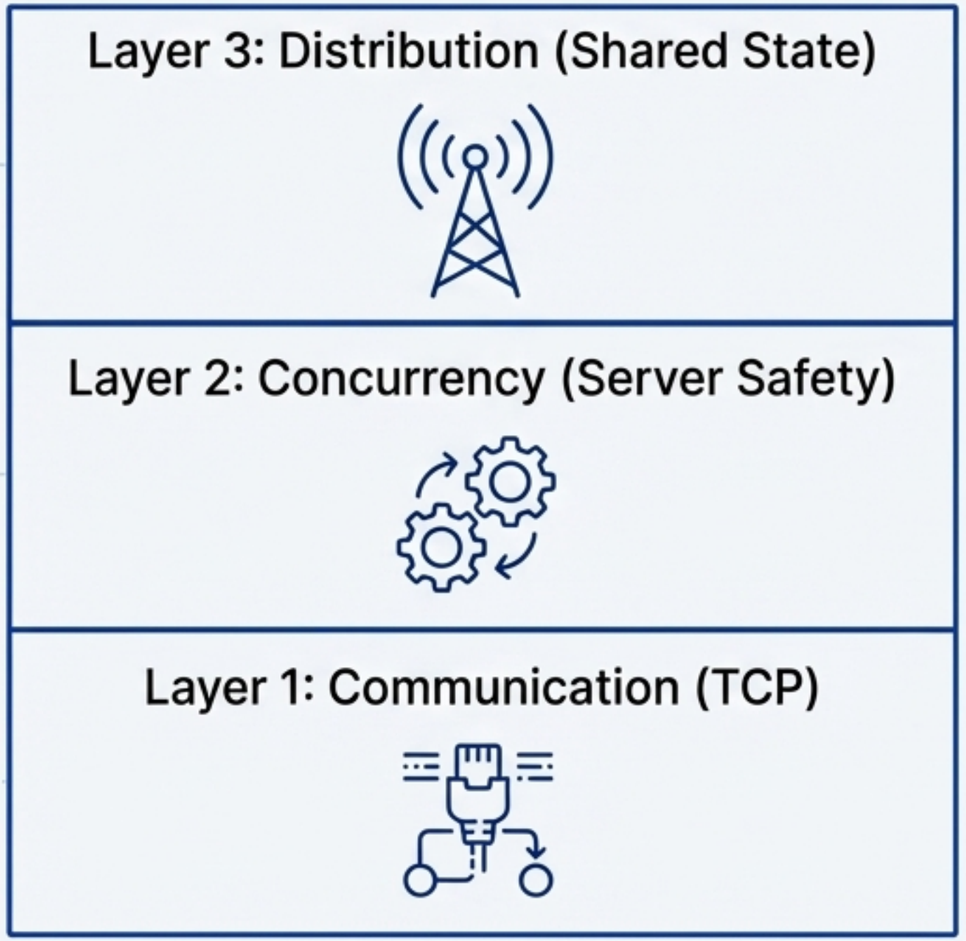
- How clients and servers talk reliably.
- **Technology**: TCP Sockets

Layer 2: The Core Logic

- How the server thinks without corrupting itself.
- **Patterns**: Concurrency & Thread Safety

Layer 3: The Shared Reality

- How we synchronize state across all users.
- **Architecture**: Server-Authoritative Broadcasting



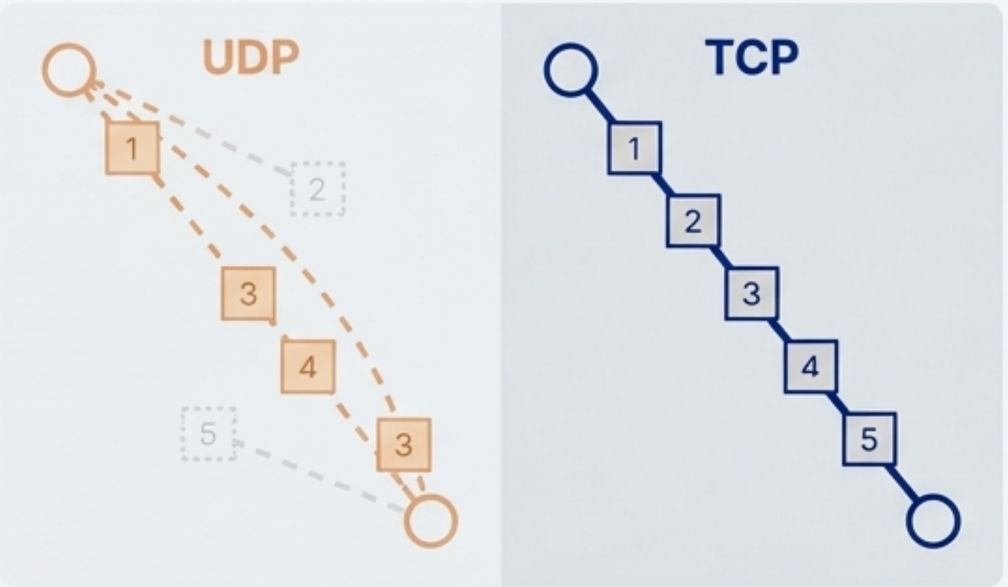
Layer 1: Forging a Reliable Link with TCP

For a collaborative app, losing a message is not an option. If a user adds a task, everyone *must* see it. We chose TCP (Transmission Control Protocol) because it provides essential guarantees that UDP (User Datagram Protocol) does not.

Protocol	Characteristics	Our Use Case
TCP	**Reliable, ordered, connection-based**	**Perfect.** We need every message, in order.
UDP	Fast, unreliable, no connection	**Unacceptable.** Good for video streaming where losing a frame is okay, but disastrous for our app.

Key TCP Guarantees

✅ **Guaranteed Delivery**: Messages won't disappear.
✅ **Guaranteed Order**: Messages arrive in the sequence they were sent.
✅ **Error Checking**: Corrupted data is automatically retransmitted.

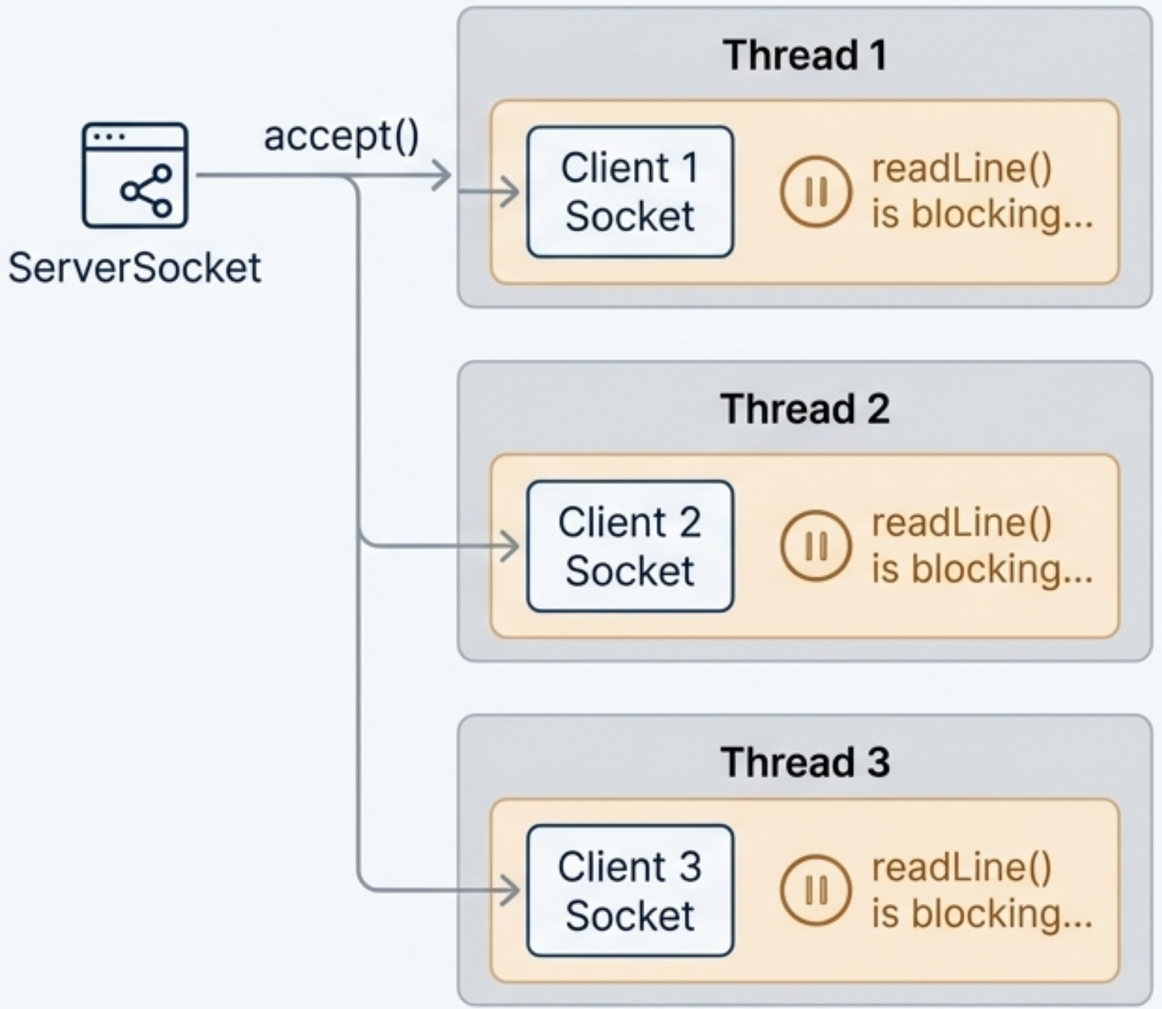


Handling Connections: The 'One Thread Per Client' Model

The server uses a `ServerSocket` to listen for new connections. When a client connects, the `serverSocket.accept()` method returns a dedicated `Socket` for that client.

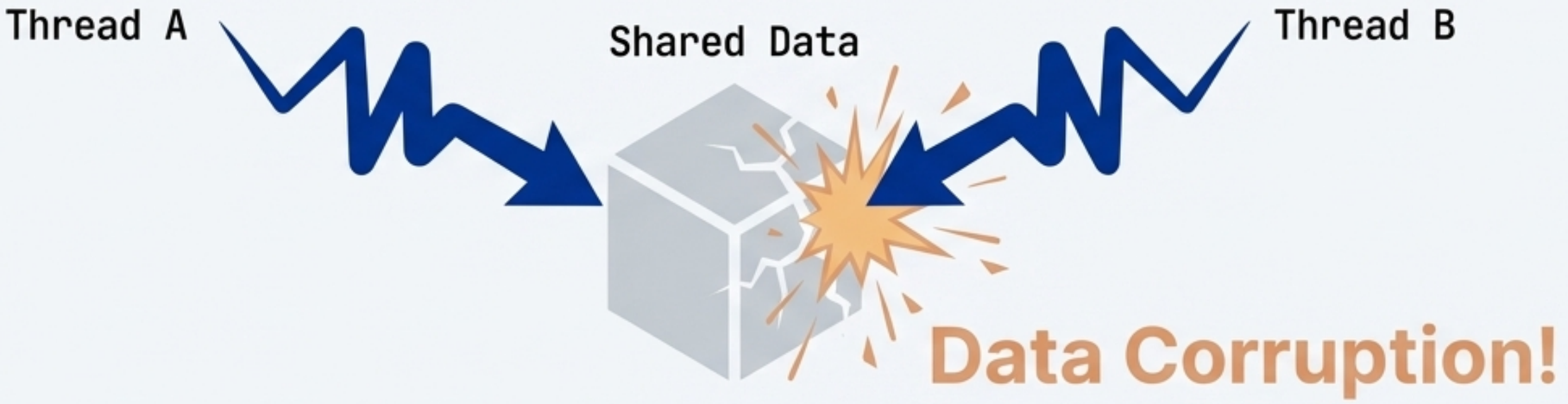
Crucially, reading from a socket (`readLine()`) is a **blocking operation**—it pauses the thread until data arrives.

- **Problem**: If we used a single thread, the entire server would freeze while waiting for one client.
- **Solution**: The server spawns a **new thread for every single client**. This allows the server to handle many clients simultaneously, even while they are idle.



Layer 2: The Race to Corrupt Data

With multiple threads running—one for each user—we now face the greatest challenge in concurrent programming: **the race condition.**



A race condition occurs when multiple threads access shared data at the same time, and at least one of them is writing. The final result depends on the **unpredictable timing of the threads.** This is the primary source of data corruption and inconsistent states.

The next few slides will break down the specific race conditions we faced and the precise patterns used to defeat them.

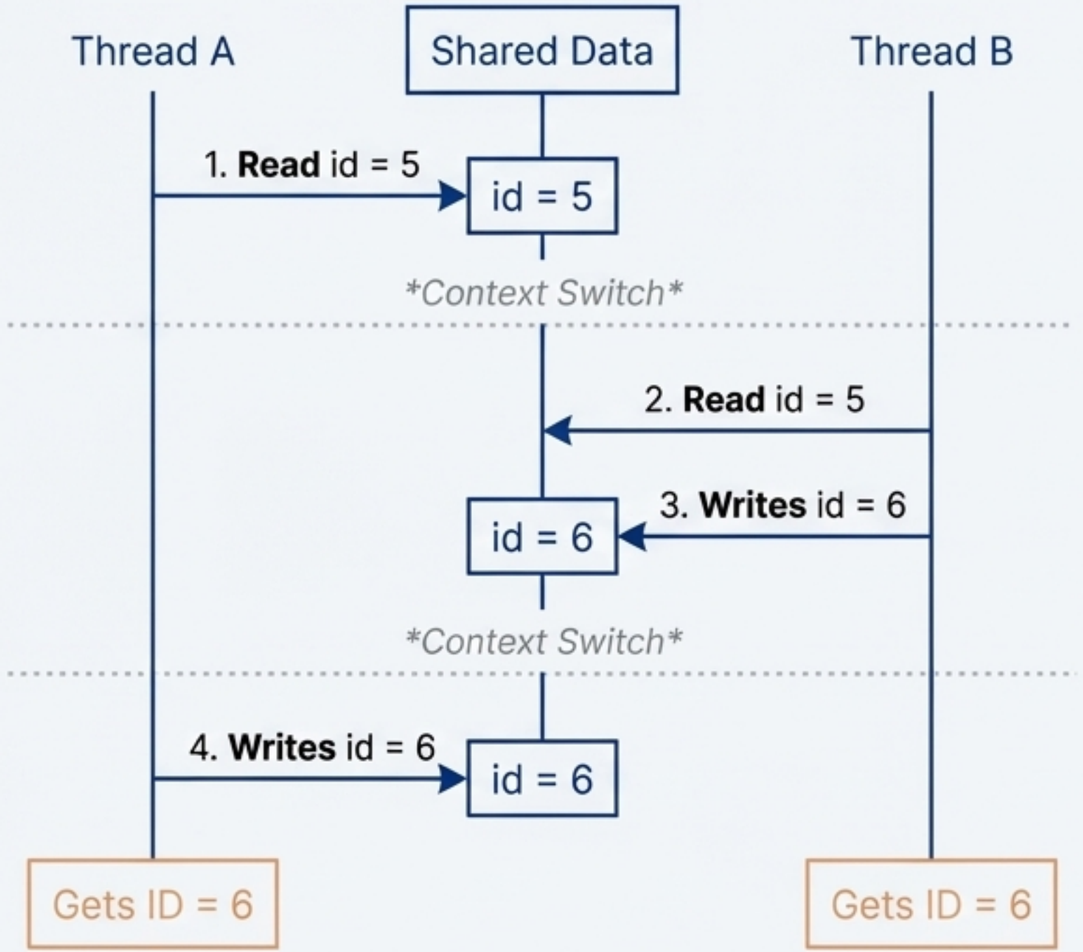
Problem #1: Duplicate IDs and the 'Lost Update'

The Scenario: Two users create a new task at the exact same time. The server needs to assign them unique IDs.

The Flaw: A simple id++ operation seems atomic, but it's actually three separate steps:

1. **Read** the current value (e.g., 5)
2. **Increment** it in memory (to 6)
3. **Write** the new value back (store 6)

Result: Both users get ID = 6. This is a 'lost update' and results in critical data corruption.



Solution: AtomicInteger for Indivisible Operations

The `java.util.concurrent.atomic.AtomicInteger` class provides a thread-safe integer where operations like incrementing are **atomic**—they happen as a single, indivisible unit. The `incrementAndGet()` method performs the read, modify, and write steps as one uninterruptible operation.

The Fix:

- Thread A calls `incrementAndGet()`. It completes the entire operation and gets `ID = 6`.
- Thread B calls `incrementAndGet()`. It starts *after* Thread A is finished, reads the new value of 6, and gets `ID = 7`.

Result: Atomicity is guaranteed. Each client receives a unique ID.

In Our Code:

Counter	Purpose
`ServerMain.clientIdCounter`	Generates unique client IDs on connection.
`TaskManager.taskIdCounter`	Generates unique task IDs on creation.

```
// Guarantees a unique ID, every time.
private final AtomicInteger taskIdCounter = new AtomicInteger(0);

public int generateNewTaskId() {
    return taskIdCounter.incrementAndGet();
}
```

Problem #2: Stolen Usernames and the 'Check-Then-Act' Flaw

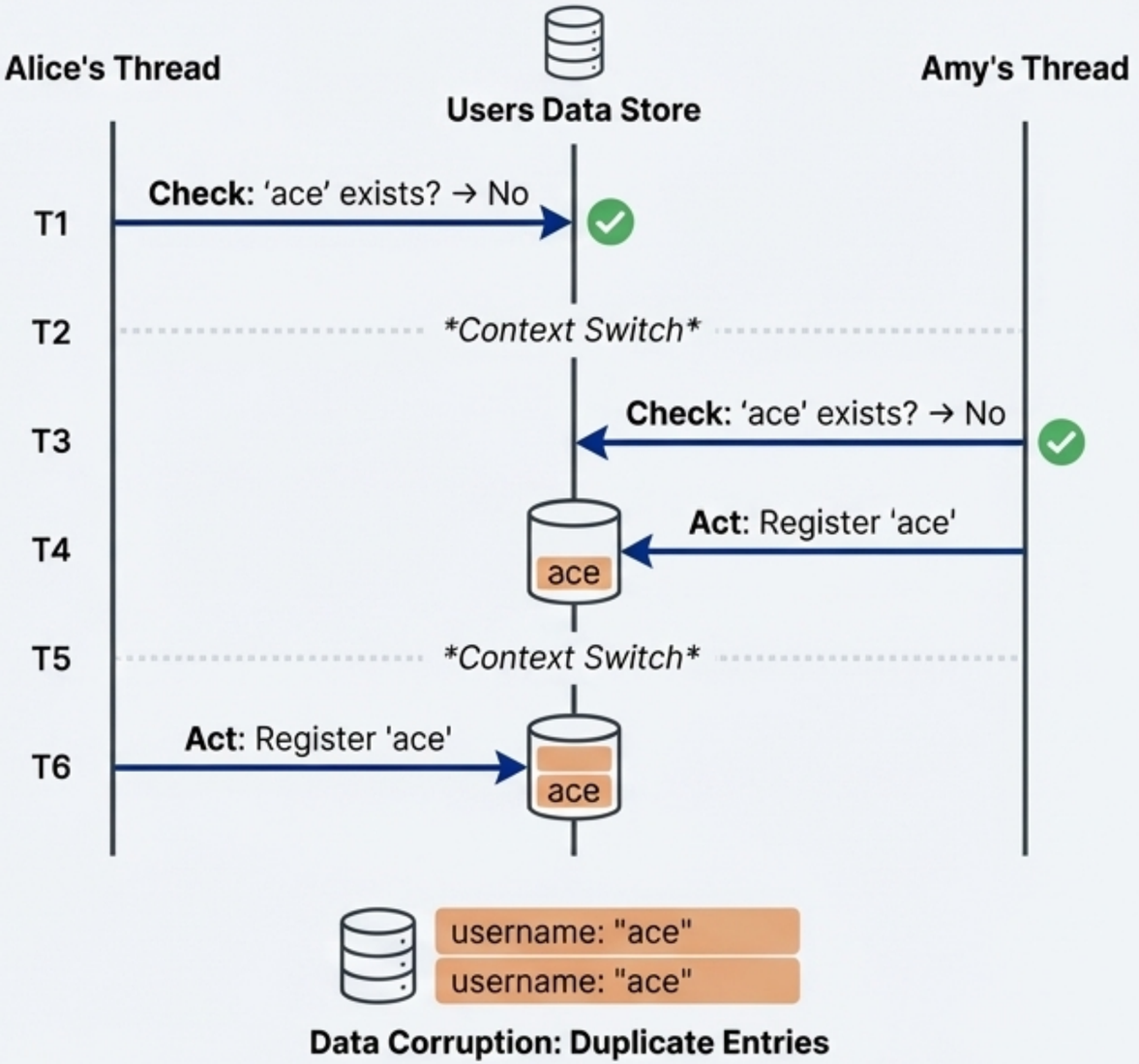
This is a classic ‘Time-of-Check to Time-of–Use" (TOCTOU) race condition.

The Scenario: Two users, Alice and Amy, both try to register the username 'ace' at the same time.

The Flaw: The server logic is dangerously split into two steps:

1. **Check**: Does the username 'ace' exist?
2. **Act**: If not, register 'ace'.

Result: Two users are logged in with the same unique username. Data is now inconsistent.



Solution: `putIfAbsent` for Atomic Check-and-Act

Thread-safe collections like `ConcurrentHashMap` provide atomic methods that combine the "check" and "act" into a single, uninterruptible operation.

The `putIfAbsent(key, value)` method does exactly what its name implies: it adds the key-value pair **only if the key is not already present**. This entire operation is atomic.

Result: Only the first thread can succeed. The atomic guarantee prevents duplicate usernames.


```
// Returns null on success, or the existing value on failure.
// This single line replaces a buggy "if(!contains) { put }" block.
Object existingUser = onlineUsers.putIfAbsent(username, clientHandler);
if (existingUser != null) {
    // Username is already taken.
}
```

Problem #3: Inconsistent States and the 'Torn Read'

The Scenario: User A is updating a task with multiple fields (e.g., description, status, modifier). At the same moment, User B tries to read that same task.

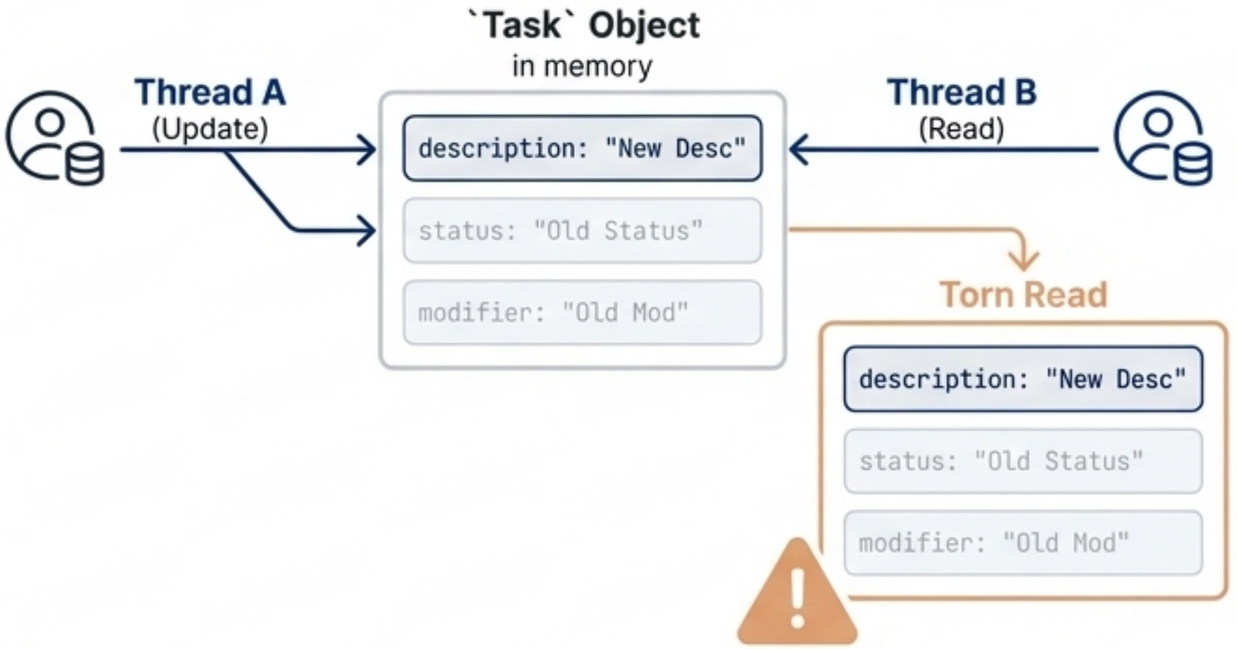
The Flaw: Without protection, Thread B can read the `Task` object *during* Thread A's update.

The Race Condition:

1. Thread A starts updating the task.
2. It successfully changes the `description`.
3. **Context Switch**
4. Thread B reads the entire `Task` object.
5. **Context Switch**
6. Thread A finishes updating the `status` and `modifier`.

Result: Thread B sees an **impossible, inconsistent state**—a "torn read":

- New Description ✅
- Old Completion Status ❌
- Old Modifier ❌



Solution: `synchronized` Blocks for Atomic Object Updates

To prevent torn reads, we must ensure that updating an object's fields is an **atomic operation**. The `synchronized` keyword provides a lock on an object, ensuring only one thread can execute that block of code on that specific object instance at a time.

- Thread A enters the `synchronized` block to update the task. It acquires a “lock” on the task object.
- Thread B attempts to read the task but is **blocked** because Thread A holds the lock.
- Thread A completes all its updates and exits the block, releasing the lock.
- Thread B can now acquire the lock and read the task.

Result: Thread B is guaranteed to see a **consistent snapshot** of the task—either *all* the old values OR *all* the new values, but never a mix.

```
public void updateTask(String newDescription, boolean newStatus, String modifier) {
    synchronized (this) { // Lock this specific task object
        this.description = newDescription;
        this.isCompleted = newStatus;
        this.lastModifiedBy = modifier;
        this.lastModifiedTime = System.currentTimeMillis();
    } // Lock is released here
}
```

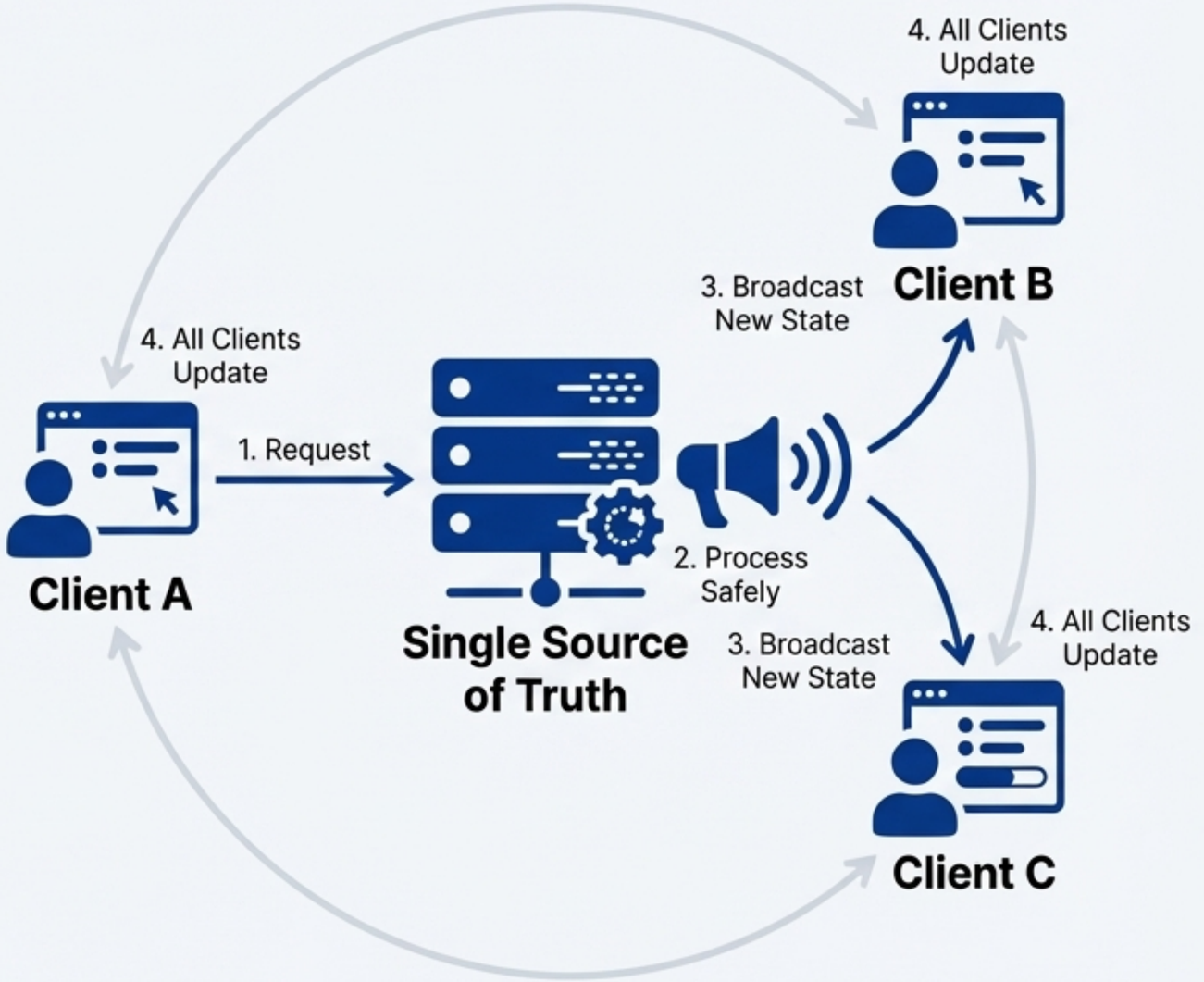
Layer 3: Broadcasting a Single Source of Truth

Now that our server is thread-safe, how do we keep all connected clients perfectly synchronized?

The Architectural Principle: Server-Authoritative Design
The server is the **single source of truth**. Clients are simple "dumb" renderers. They never modify their own state directly.

The Flow:

1. A client sends a **request** to the server (e.g., "add task 'Buy milk'").
2. The client's UI **waits**. It does not add the task yet.
3. The server processes the request, using the concurrency patterns we've discussed to ensure safety.
4. The server then **broadcasts** the official new state to **every single connected client**.
5. All clients (including the one that made the request) receive the broadcast and update their UI accordingly.



The Power of Patience: Why Clients Wait for the Broadcast

A critical detail of this architecture is the **rejection of "optimistic updates."** The client that initiates an action does *not* immediately update its own UI. It waits for the server's confirmation broadcast, just like every other client.

Benefit	Explanation
Absolute Consistency	All clients see identical data because they all render from the same broadcast stream.
Server Authority	The server is solely responsible for assigning IDs and validating data. The client doesn't know the new Task ID until the server says so.
Graceful Failure	If the server is down or the request fails, the task simply never appears. No need to 'roll back' a fake UI update.
Simpler Client Logic	There is only one code path for updating the UI: `handleServerMessage()`. This dramatically reduces client-side complexity.

Client-Side Code Example (Android):

```
// The user clicks "add task"
fun onAddTaskClicked(description: String) {
    // We ONLY send a message to the server.
    // We DO NOT update the local task list here.
    serverApi.send("ADD:$description")
}
```

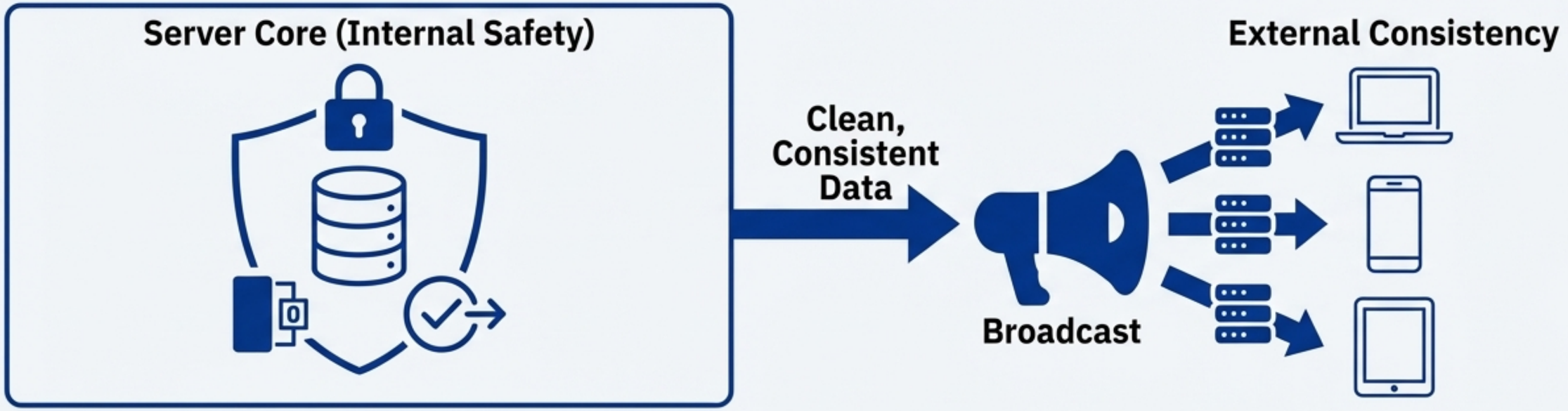
Action sent, UI waits.

```
// The UI updates ONLY when the broadcast arrives
private fun handleServerMessage(message: String) {
    if (message.startsWith("TASK_ADDED")) {
        // Now we parse the official task data and update the UI.
        val task = parseTask(message)
        adapter.addTask(task)
    }
}
```

Official state from server updates UI.

How Concurrency Enables Distribution

The distributed broadcast architecture is powerful, but it would be useless without a thread-safe foundation. The concurrency patterns we implemented are what guarantee the integrity of the data that gets broadcast to all users.



Internal Safety Creates External Consistency

When This Distributed Action Happens...	...This Concurrency Pattern Protects It Internally
Two users add tasks at the same time	`AtomicInteger` ensures they get unique IDs.
Two users try to register the same username	`putIfAbsent()` ensures only one succeeds.
A user reads a task while another updates it	`synchronized` blocks prevent a 'torn read' from being broadcast.
The server broadcasts to all clients while one disconnects	`ConcurrentHashMap` allows for safe iteration and removal.

The Complete Architectural Summary

Together, these three layers—Networking, Concurrency, and Distribution—create a reliable, real-time, and collaborative application.

Concurrency Patterns

Pattern	Problem It Solves
AtomicInteger	Prevents "lost updates" in counters (e.g., unique IDs).
Atomic Check-and-Act	Prevents TOCTOU race conditions (e.g., unique usernames).
Synchronized Updates	Prevents "torn reads" on multi-field objects.

Distributed System Concepts

Concept	Purpose
Server-Authoritative	Ensures all clients see the same data from a single source of truth.
Broadcasting	Provides real-time synchronization to all connected clients.
No Optimistic Updates	Simplifies client logic and prevents false success states.

TCP Networking Foundation

Concept	Purpose
TCP Sockets	Provide reliable, ordered communication channels.
One Thread Per Client	Allows the server to handle many simultaneous blocking connections.