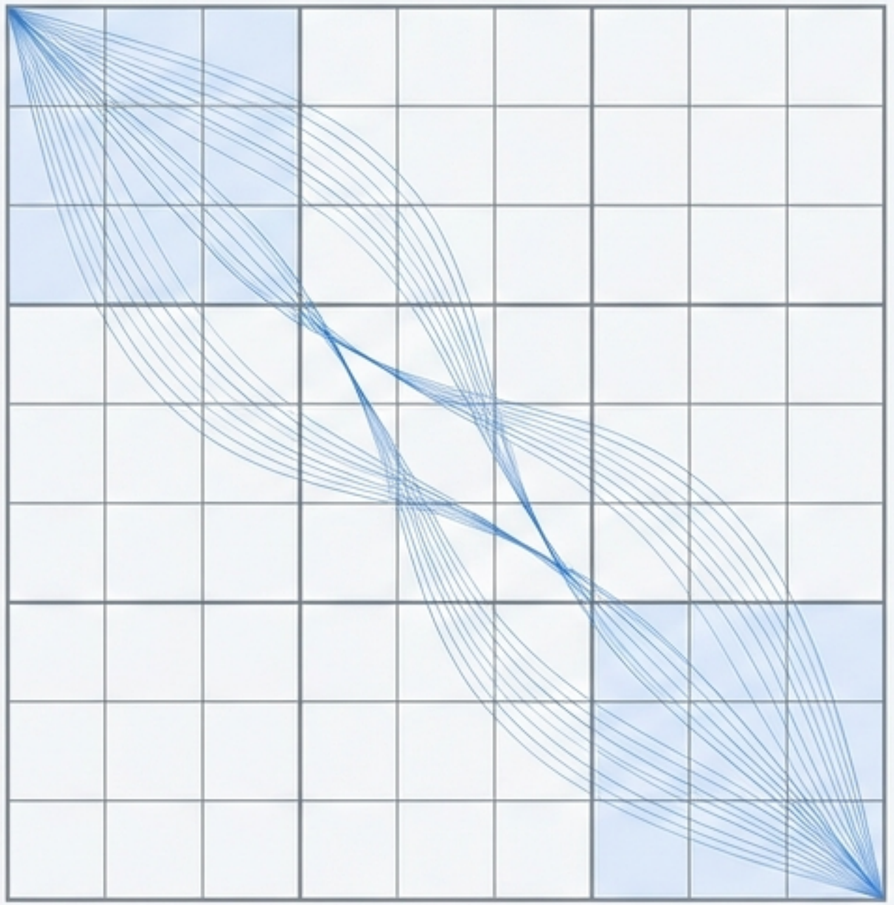
Anatomy of a Concurrent Sudoku Server

A Technical Deep Dive into Fine-Grained Locking and Real-Time State Management



The Core Challenge: Real-Time Shared State

At the heart of any real-time multiplayer game is a shared, mutable state—in our case, the Sudoku board.

What happens when two players try to modify the same piece of data at the same microsecond?

This leads to a classic **Race Condition**, where the final state of the system depends on the unpredictable sequence of thread execution.

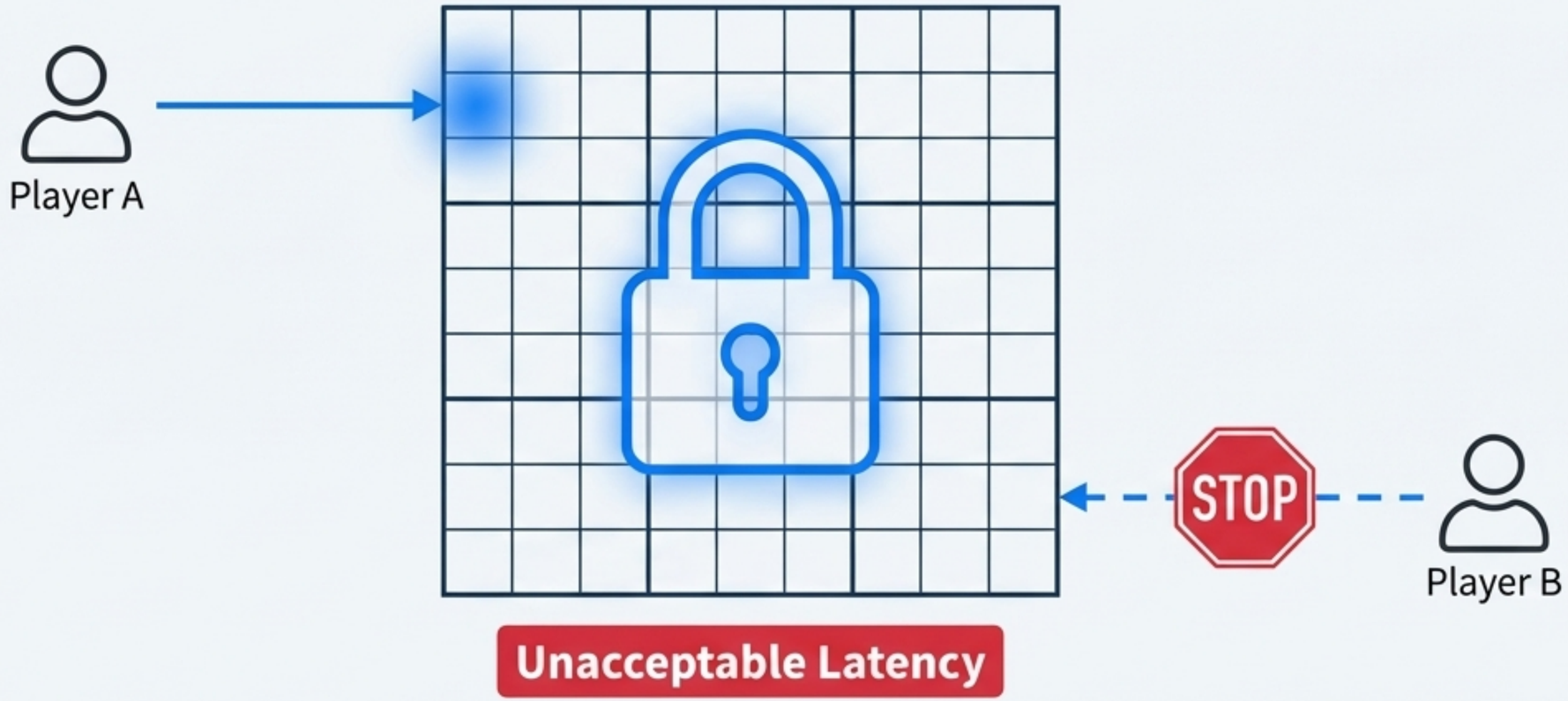
A common outcome is the **“Lost Update”** problem: one player’s move is silently overwritten by another’s, leading to data corruption.

The Naive Approach: Coarse-Grained Locking

The simplest way to prevent race conditions is to lock the entire `Game` object for every single move. This ensures only one player can act at a time.

While this guarantees data integrity, it creates an unacceptable user experience by serializing all actions.

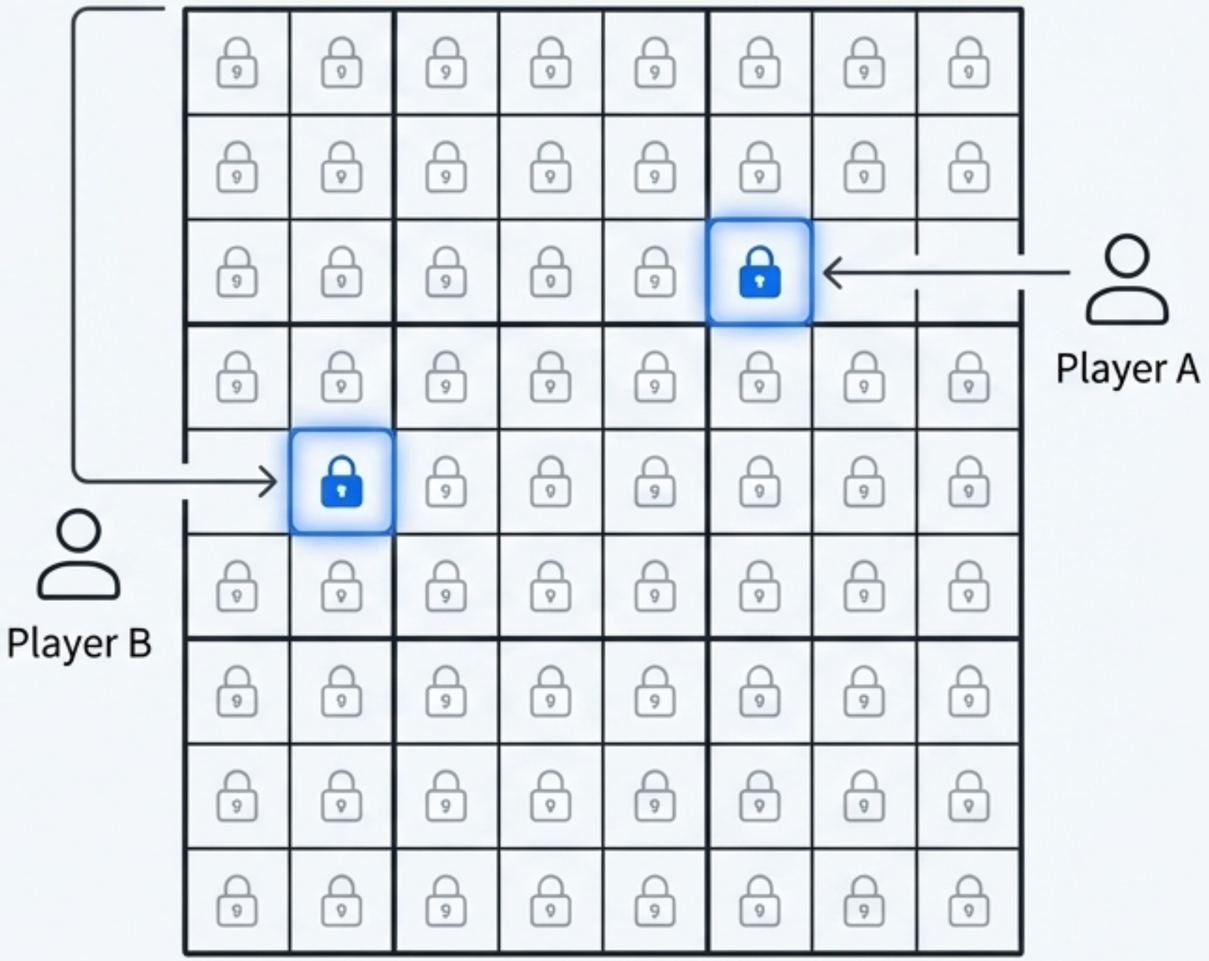
```
// Hypothetical approach with a single global lock
public synchronized MoveResult processMove(...) {
    // Only one player can execute this method
    // at a time for this entire game instance.
    // ... game logic ...
}
```



The Core Solution: Fine-Grained, Cell-Level Locking

Instead of one lock for the entire game, we create 81 individual locks—one for each cell on the Sudoku board. This allows players to make moves on different cells completely in parallel, maximizing concurrency and eliminating unnecessary blocking.

```
public class Game {
  // ... other fields
  private final Lock[][] cellLocks; // 9x9 grid of locks!

  public Game(String gameCode, String difficulty) {
    // Initialize 9x9 grid of locks
    this.cellLocks = new Lock[9][9];
    for (int i = 0; i < 9; i++) {
      for (int j = 0; j < 9; j++) {
        cellLocks[i][j] = new ReentrantLock();
      }
  }
}
```



Implementation Deep Dive: Non-Blocking with `tryLock()`

Using a standard `lock()` call would cause a player's client to freeze if they tried to click on a cell another player was currently editing. Instead, we use the non-blocking `tryLock()`. It attempts to acquire the lock and returns `false` immediately if it can't, allowing us to provide instant feedback to the user ("Cell is busy").

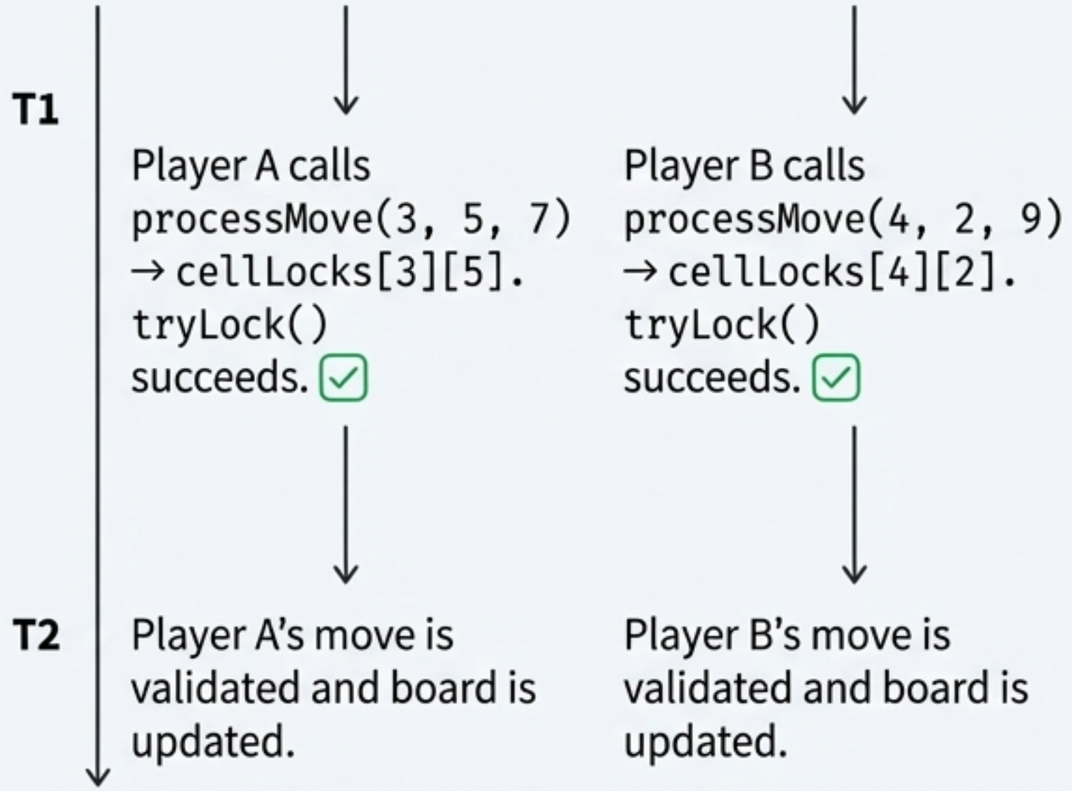
```
// NON-BLOCKING ATTEMPT
Lock lock = cellLocks[row][col];
if (!lock.tryLock()) {
    return new MoveResult(..., false, "Cell is busy");
}

try {
    // CRITICAL SECTION
    // Only one thread can be here for this specific cell.
    // ... update board, check solution, update score ...
} finally {
    lock.unlock();
}
```

Returns `false` immediately if the lock is held. No blocking, no UI freeze.

Guarantees the lock is always released, even if an error occurs. Prevents deadlocks.

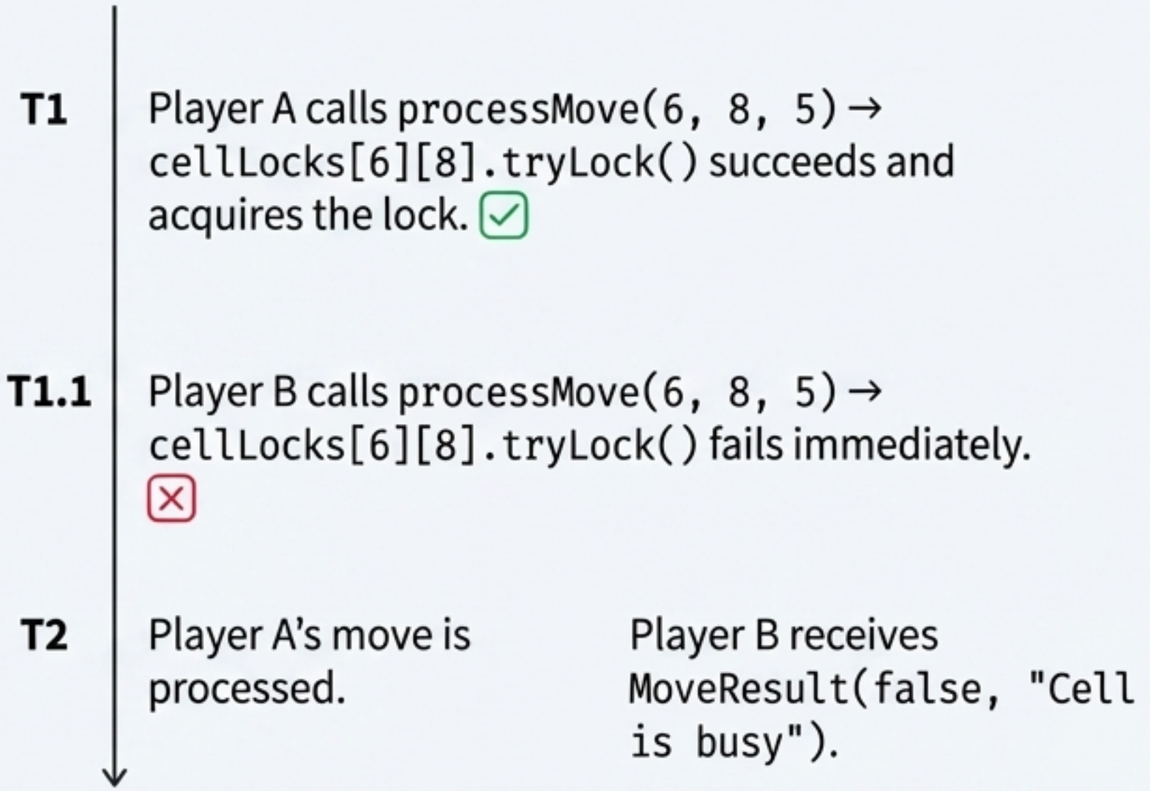
Validating the Model: A Tale of Two Moves

Scenario 1: Different Cells (Concurrent Success)



Result: Both moves succeed in parallel.

Scenario 2: Same Cell (Conflict Handled Gracefully)



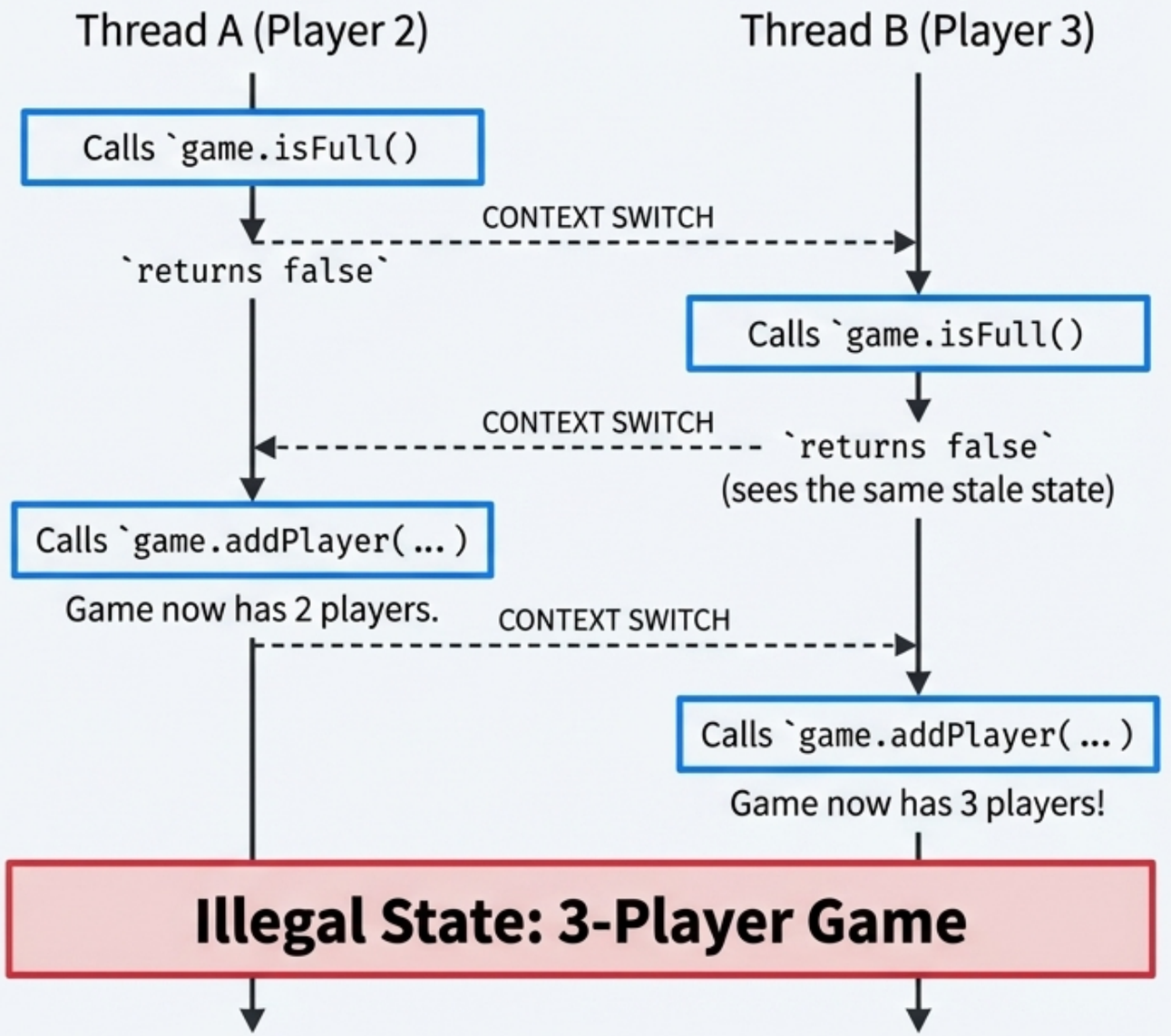
Result: Alice succeeds, Bob gets immediate feedback.

The Second Challenge: Atomic Matchmaking

Game state isn't the only shared resource. The list of available games in the GameManager also requires protection.

Consider a game with one slot left. What happens if two players try to join at the same time?

This creates a **"Check-Then-Act" race condition**.



The Right Tool for the Job: `synchronized` & `ConcurrentHashMap`

Different problems demand different concurrency strategies. A single lock is inefficient for gameplay but perfect for a short, critical transaction like joining a game.

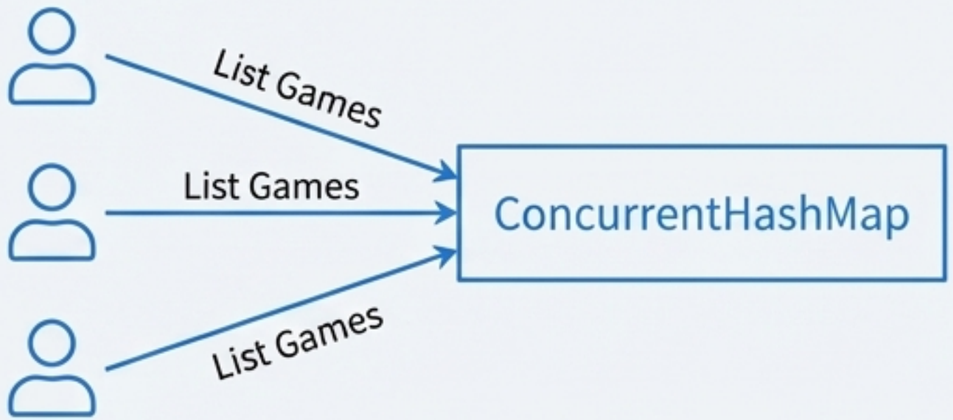
Solution 1: Atomic "Check-Then-Act"

The `synchronized` keyword on the `addPlayer` method ensures the check for an open slot and the act of adding a player happen as a single, indivisible (atomic) operation.

```
public synchronized int addPlayer(String playerName,
WebSocket session) {
    if (player2 == null) {
        player2 = new Player(2, playerName, session);
        gameStarted = true;
        return 2;
    }
    return -1; // Game is full
}
```

Solution 2: High-Performance Reads

`GameManager` uses `ConcurrentHashMap` to store game instances. This collection is highly optimized for reads, allowing many players to list available available games simultaneously without any blocking.

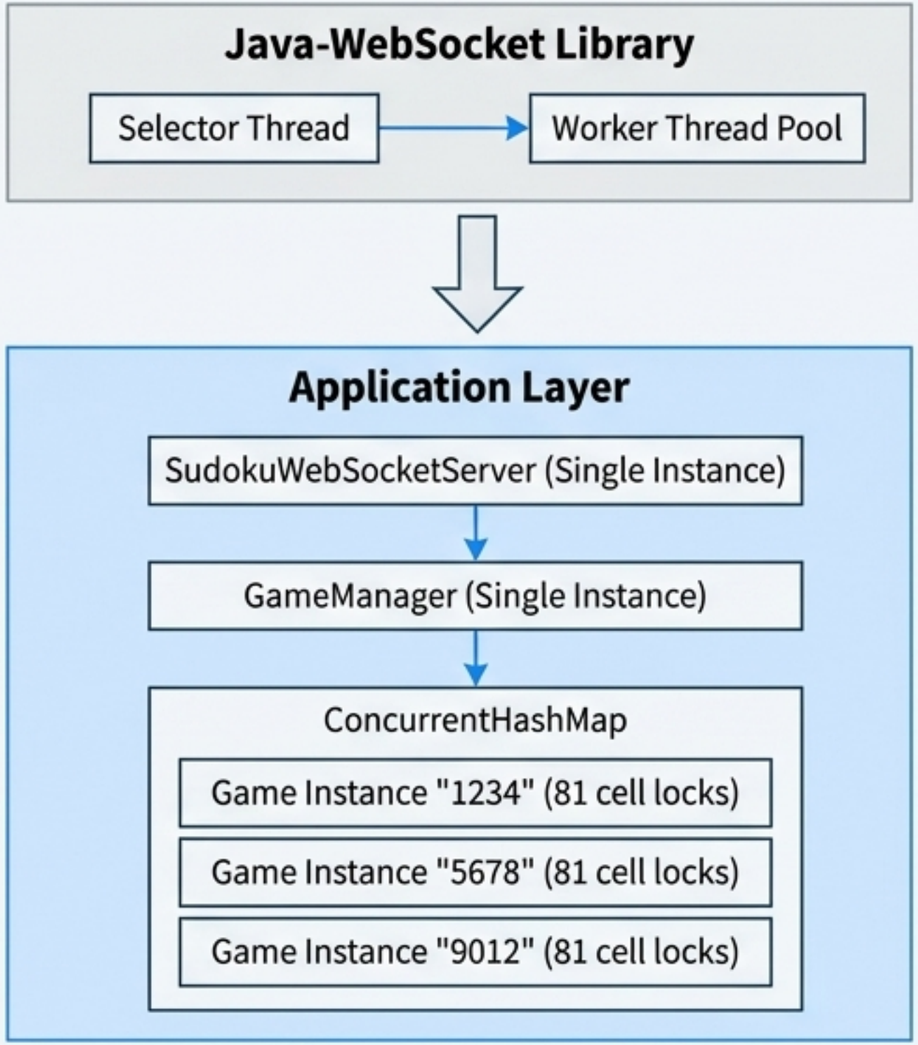


The Engine Room: Server Architecture & Threading Model

The server is built on the `Java-WebSocket` library, which manages connections and dispatches work to a thread pool.

The library uses a cached thread pool, meaning that `onMessage` calls for different clients can, and will, execute concurrently on separate worker threads.

This is the fundamental reason our application logic—from the `GameManager` down to the individual `Game` instances—must be meticulously thread-safe.



Full Execution Flow: From Connection to Completion

1. **Client Connects**: `onOpen` is triggered on a worker thread.

2. **Client Sends `CreateGameRequest`**: A worker thread executes `onMessage`.

3. **`GameManager` Creates Game** : A new `Game` instance (with 81 locks) is created and stored in the `ConcurrentHashMap`.

4. **Second Player Joins**: A different worker thread handles the `JoinGameRequest`. `Game.addPlayer()` is called within a `synchronized` block to prevent a race condition.

5. **Concurrent Moves**: Multiple `processMove` calls execute in parallel on different worker threads, using fine-grained `tryLock()` on individual cells to manage state. A collision on the same cell results in an immediate “Cell is busy” response.

6. **Game Ends**: The game is marked as complete.

7. **Cleanup**: The `GameManager` removes the finished game from the `ConcurrentHashMap`.

A Note on the Client: Maintaining Responsiveness

Server-side non-blocking design is only half the battle. The Android client must also handle network I/O without freezing the UI.

Kotlin Coroutines are used to manage asynchronous operations.

A WebSocket listener runs in a coroutine on a background `IO` dispatcher, processing incoming messages off the main UI thread.

Messages are emitted into a thread-safe `SharedFlow`, which the UI collects safely to update the game board.

```
// Runs on a background thread, never
blocking the UI
client.webSocket(...) {
   for (frame in incoming) { // Iterates over
WebSocket frames
      if (frame is Frame.Text) {
         // Decode JSON on a background thread
         val message =
json.decodeFromString(frame.readText())
         // Safely emit to the UI layer
         _messages.emit(message)
      }
   }
}
```

Key Architectural Takeaways

- **Fine-Grained Locking (`ReentrantLock`):** The essential pattern for maximizing concurrency and throughput on core, contended data structures like the game board.
- **Optimistic, Non-Blocking `tryLock()`':** The key to providing responsive, real-time user feedback instead of UI freezes during contention.
- **Atomic Operations (`synchronized`):** The right tool for safeguarding short, multi-step transactions like joining a game, preventing "check-then-act" race conditions.
- **Thread-Safe Collections (`ConcurrentHashMap`):** The solution for managing shared collections that require high-performance, non-blocking reads.

Core Principle: Match the concurrency strategy to the specific data contention problem. There is no one-size-fits-all solution.

Q & A

https://github.com/berkaybilen/cmpe436-sudoku-project