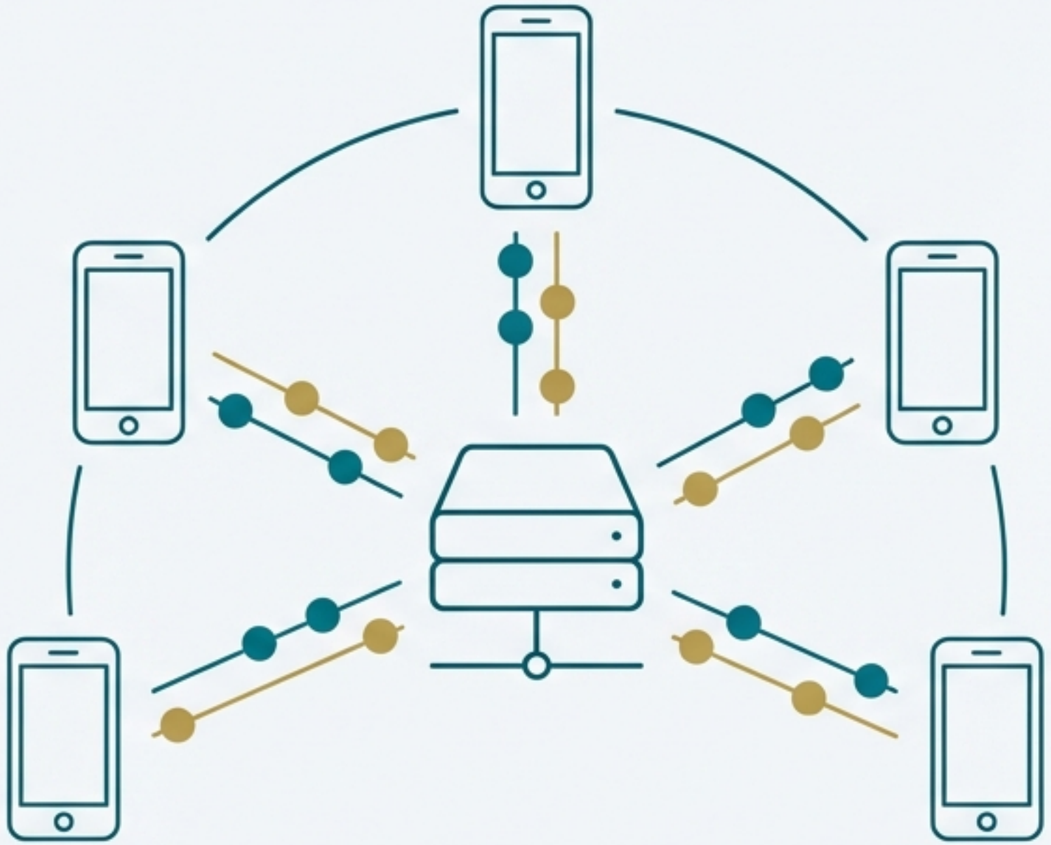
SetNet: A Full-Stack Deep Dive into a Real-Time Multiplayer Game

An analysis of architecture, concurrency, and custom UI development.

NotebookLM

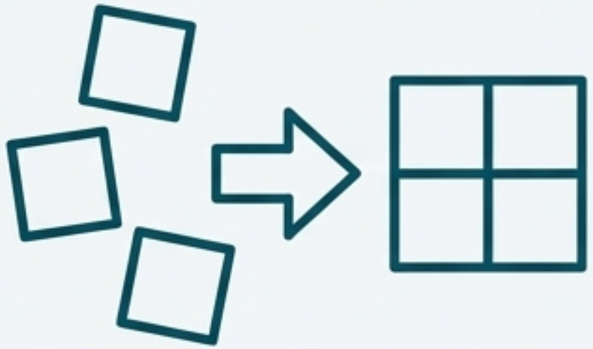
The Core Challenge: Keeping Everyone in Sync

In a connected world, how do we build applications where multiple users can interact with a shared state simultaneously and seamlessly? A simple action by one user must be instantly and accurately reflected for all others. This project, SetNet, is a practical, ground-up implementation of a solution to this problem, demonstrated through the logic of a real-time multiplayer card game.



NotebookLM

Deconstructing the Problem: The Three Technical Hurdles



State Synchronization

How do we ensure every player sees the exact same game board at the same time? The server must be the single, authoritative source of truth.

Concurrency & Race Conditions

What happens when two players try to claim the same set of cards at the exact same moment? The system must process inputs fairly and prevent conflicts.

Custom UI Rendering

How do you efficiently draw a complex, interactive game state on a constrained mobile screen without relying on a pre-built game engine?

NotebookLM

The Solution: An Overview of SetNet

Centralized Java Server

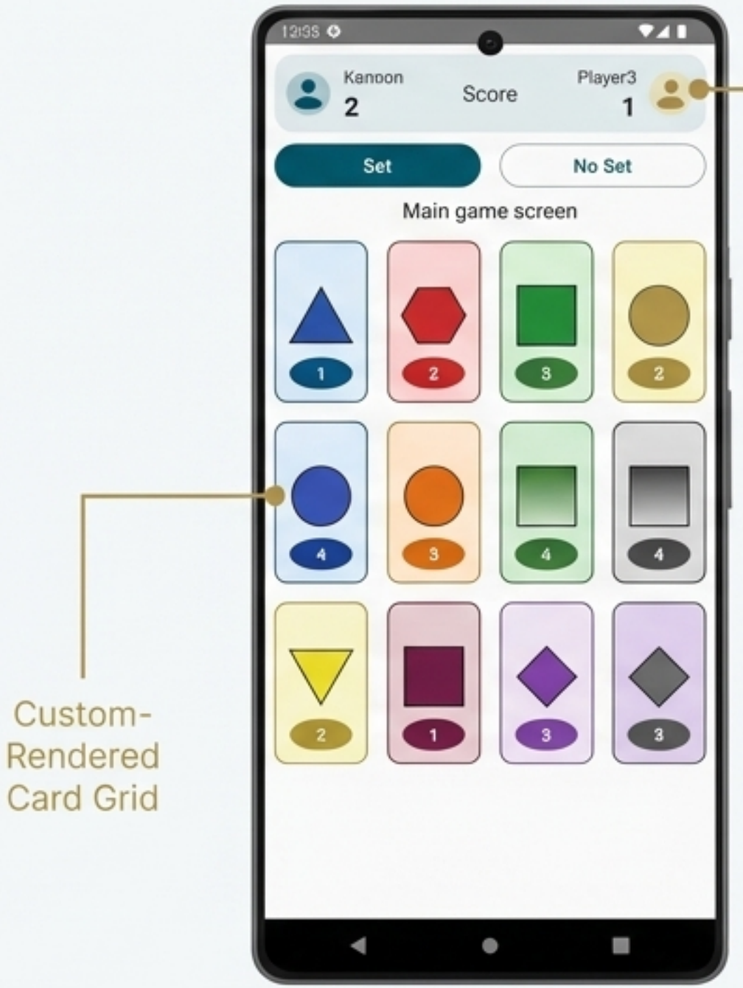
A single source of truth manages the deck, board, and player moves. It runs a dedicated thread for each connected client, ensuring responsive, independent communication.

Robust Android Client

A native application provides a fluid, responsive user experience. It handles user input and renders the game state received from the server.

Real-Time TCP Networking

A lightweight, custom text-based protocol facilitates instant communication for moves, board updates, and score changes between the server and all clients.



NotebookLM

System Architecture: The Client-Server Model

Backend (Java Server)

`SetServer.java`

- Game Logic
- Concurrency Control
- Network Host

`ClientHandler.java`

- Per-Player Thread
- I/O Management

Frontend (Android Client)

`MainActivity.java`

- UI Logic
- Event Handling

`SetCardView.java`

- Custom `onDraw` Rendering

`NetworkClient.java`

- Background Networking
- UI Thread Updates via `Handler`

`MOVE:id1,id2,id3`

`BOARD:[data]`, `SCORE:pid:pts`

NotebookLM

Architecture Deep Dive: The Java Backend

Authoritative State Management

The server maintains the master game state in `Collections.synchronizedList(board)` and the `deck`. This prevents direct state manipulation by clients.

Concurrent-Safe Collections

To handle simultaneous updates from multiple player threads, the implementation avoids standard HashMaps. Instead, it leverages thread-safe collections for critical data structures:

```
// From SetServer.java
private static final List<ClientHandler> players = new CopyOnWriteArrayList<>();
private static final Map<Integer, Integer> playerScores = new ConcurrentHashMap<>();
```

`CopyOnWriteArrayList` is ideal here; the player list is read frequently by all threads but modified rarely (only on connect/disconnect), optimizing for read-heavy concurrency.

Allows for atomic, lock-free score updates from multiple player threads simultaneously.

NotebookLM

Architecture Deep Dive: The Android Frontend

Custom View Rendering

Instead of using static images, the `SetCardView` class draws each card's unique combination of shapes, shadings, and colors from scratch using Android's low-level `Canvas` and `Paint` APIs. This provides maximum flexibility and performance.

```
// From SetCardView.java onDraw() method
if (card.shading == 0) { // SOLID
    paint.setStyle(Paint.Style.FILL);
    canvas.drawPath(path, paint);
} else if (card.shading == 1) { // STRIPED
    // ... logic for striped fill ...
} else { // OPEN
    paint.setStyle(Paint.Style.STROKE);
    canvas.drawPath(path, paint);
}
```

Asynchronous UI Updates

`NetworkClient.java` runs on a background thread to prevent network latency from freezing the user interface. It uses a `Handler` to safely post data received from the server (like board updates) back to the main UI thread for rendering.

NotebookLM

Key Technical Challenge: Preventing Race Conditions

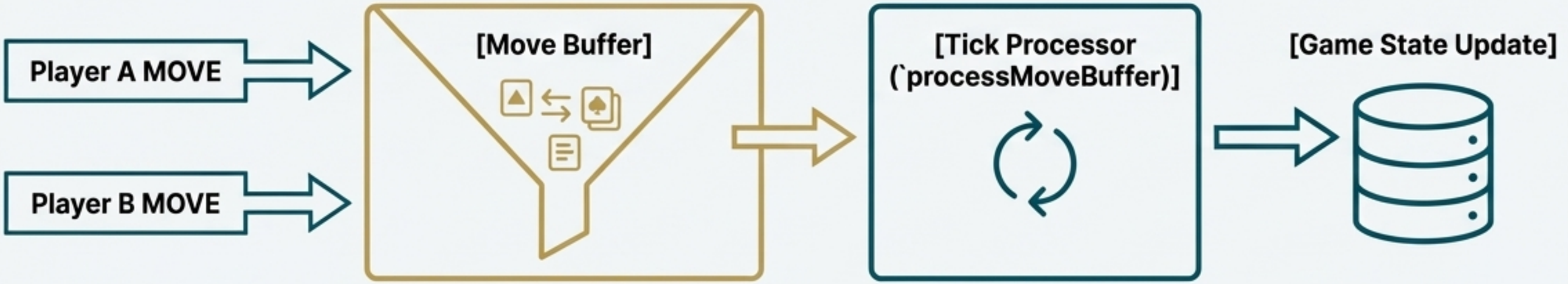
The Problem

Two players see a 'Set' and submit their moves almost simultaneously. Their `MOVE` requests arrive at the server milliseconds apart. How do we prevent the same cards from being claimed twice and ensure fairness?

The Solution: The Tick & Buffer System

The server doesn't process moves instantly. Instead, it uses a time-based buffer:

1. A `tickLoop` runs every 500ms.
2. All incoming `MOVE` requests within that 500ms window are added to a synchronized `moveBuffer`.
3. At the end of the tick, the `processMoveBuffer()` method locks the buffer and processes all queued moves atomically, checking for conflicts and updating the game state just once.



The critical logic is protected by `synchronized (moveBuffer) { ... }` from `SetServer.java`.

NotebookLM

The Custom Network Protocol

Communication is handled via a simple, human-readable, text-based protocol over TCP. This avoids the overhead of more complex serialization formats for a project of this scope.

Client to Server Commands	Server to Client Commands
`NAME:PlayerName`	`BOARD:1-0-1-2-0,…`
`MOVE:id1,id2,id3`	`SCORE:playerId:points`
	`NAMES:1-Alice,2-Bob`
	`MSG:Welcome Player 1`
	`WIN:playerId`
	`TICK:1234`

NotebookLM

Future Improvements & Learnings

Protocol Upgrade

Migrate from the simple string-based protocol to a more robust and extensible format like JSON or Protocol Buffers. This would improve improve error handling and make adding new message types easier.

Enhanced Client UI/UX

Implement animations for card transitions when a set is found and removed. Add a more polished ‘Game Over’ ‘screen and a lobby system for waiting for other players.

Reconnection Logic

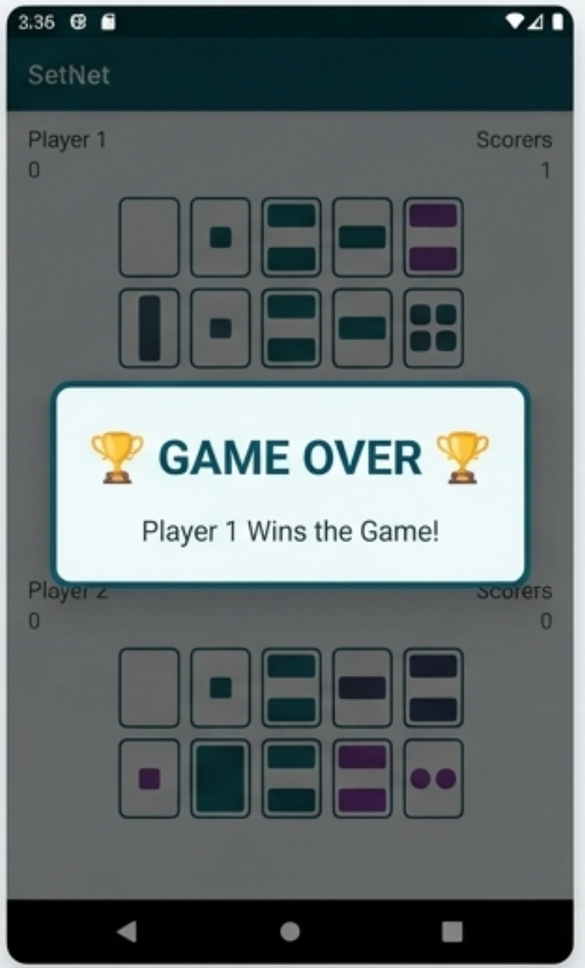
Currently, a dropped connection means the player is out. Implement a session token system to allow clients to rejoin a game in progress if their network connection is temporarily interrupted.

NotebookLM

Conclusion: A Complete Full-Stack System

SetNet is more than a game; it is a functional, end-to-end demonstration of solving core distributed systems challenges. It successfully manages concurrent state, processes user input fairly, and delivers a responsive experience on a custom-built native client. The project showcases concrete proficiency in:

- **Java Concurrency**: Using thread-safe collections and synchronized logic to prevent race conditions.
- **Custom Android UI Development**: Building complex, interactive views from first principles.
- **Client-Server Architecture**: Designing and implementing a complete, real-time networking model.



NotebookLM