**COIT13329**

**Applied Distributed Systems**

**Assessment 2**

**By: Mansi Bhardwaj - 12299442**

**Professor Jamie Shield**

**Central University Queensland (Brisbane)**

**8 June 2025**

**1. Requirements Analysis**

The main objective of this distributed system is to create a multiplayer version of the popular Towers of Hanoi puzzle that can be accessed and played by multiple users across different systems. The idea is to allow users to join the same game sessions remotely, to either play together with others or to compete against other users to solve the puzzle as fast as possible. By using a distributed architecture, the system will have real-time interaction between the players, manage shared game state across different nodes, and make sure that the game logic remains consistent even if there is ever any failure such as network delays or node failures. In the long term, the system could also support more features like matching players with similar levels, leaderboards, and accessible globally over the internet to possibly a billion people.

**Scenario 1 Organisation: a large group (100) of people from the same company**

A centralised client-server architecture is best for this scenario. The system will have a central game server (or a small cluster) managing all game logic and state, with clients accessing the game through a web-based front-end. This setup makes the most sense for a secure, LAN-based environment and simplifies maintenance, security, and performance. When designing a distributed system for 100 users in the same company, here are some requirements to keep in mind:

### ***Functional Requirements***

* Users should be able to log in and join game sessions easily. (using organisation credentials)
* Users can create or join game rooms and play the Towers of Hanoi game with others via the company network (LAN/VPN).
* The game state should update in real-time so all players see the same moves.
* There should be a way for players to communicate, like a chat or notifications system.
* Game progress needs to be saved so players can resume if they disconnect or the system crashes.
* Interface should be simple and usable.

### ***Non-Functional Requirements***

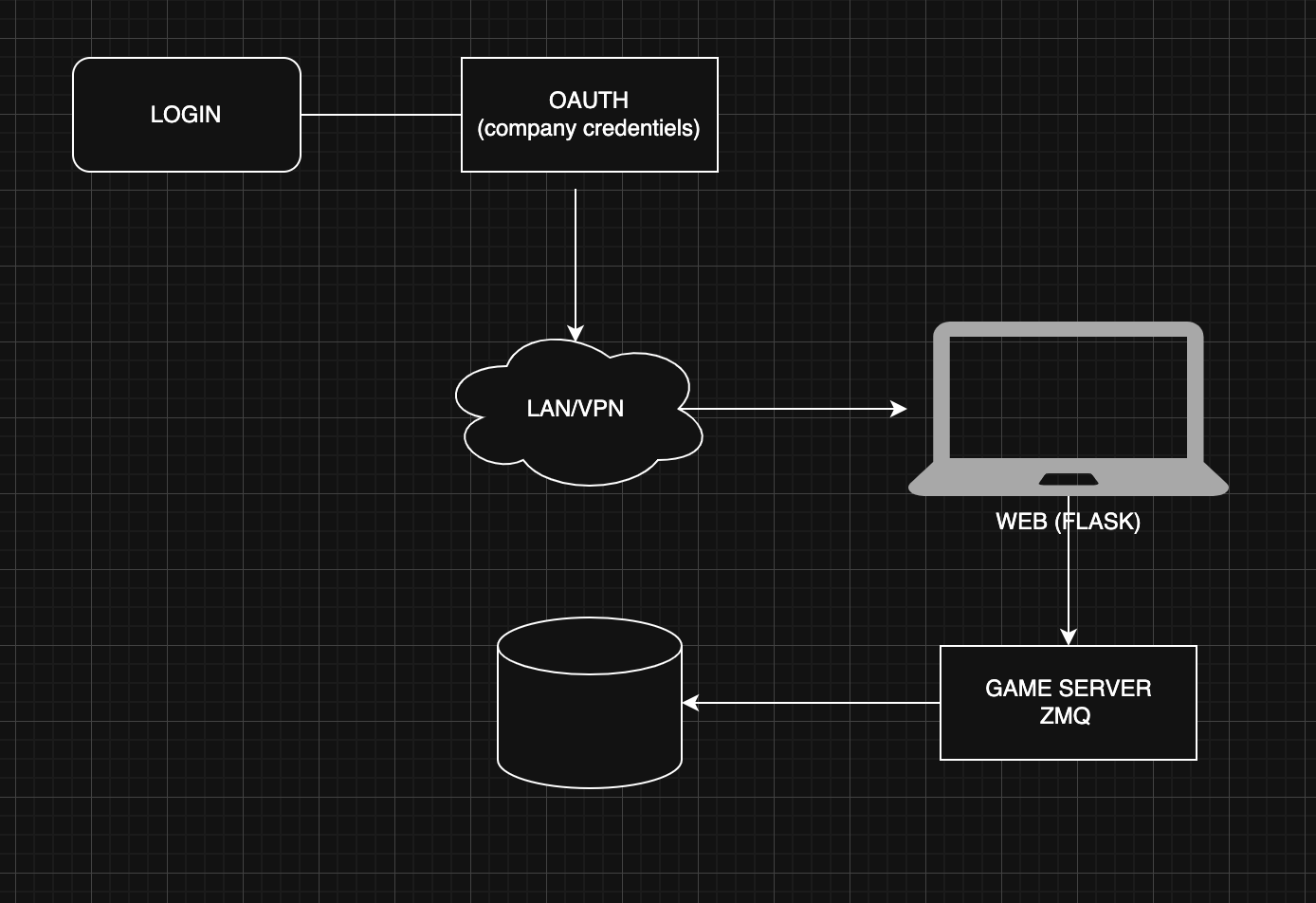
* The system should be able to handle 100 users playing at the same time across many different games.
* Security should be included to make sure only company users can access the game.
* If a server fails, the game keeps running without losing progress.
* Updates of the game should happen quickly.

***Fault Tolerance***

* If a player disconnects, they should be able to rejoin without restarting the game.
* Failed messages should be retried automatically or queued.
* The system should be able to recover from a node/server crash without affecting all users.
* Invalid moves or game interruptions should be handled.

***Technology and Design***

* ZeroMQ for messaging between game servers.
* Using Docker or virtual machines hosted on the company’s intranet.
* A web-based front-end with Flask and Socket.IO for real-time updates.
* Using tools like ZooKeeper to manage server leadership and synchronization.
* A relational DB (PostgreSQL or SQLite) for user info, game histories, user levels and points and saved states.

****

**Scenario 2 Internet: 1 billion people from across the world.**

Since this system is meant for a huge number of users around the world (up to 1 billion), a good approach would be to use a cloud-based, decentralised architecture that’s built using microservices. This setup makes it easier to scale the system and helps it be reliable even in case of failures. Each part of the system can be developed and run separately. These services would be deployed using cloud platforms like **AWS**, **Google Cloud**, or **Azure**, and spread across different regions to make sure users everywhere around the globe can access the system.

***Functional Requirements***

* Users should be able to log in and join game sessions easily.
* Users should be able to create or join public and private game sessions with players around the world.
* The game state should update in real-time so all players see the same moves.
* There should be a way for players to communicate, like a chat or notifications system.
* Players should be matched with others based on skill level or region.
* Should support multiple languages so users can have global access.

***Non-Functional Requirements***

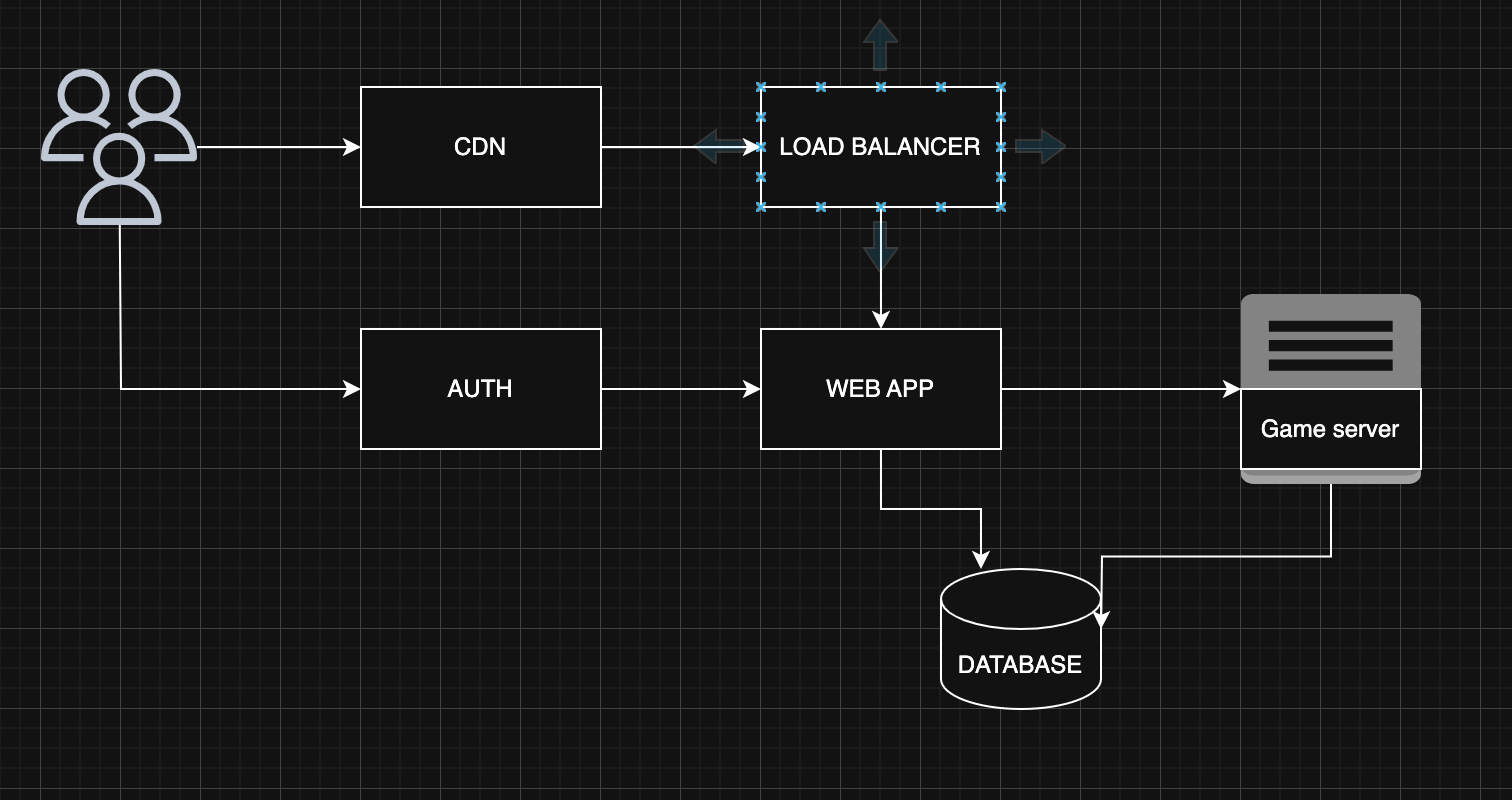
* System should be able to scale up or down dynamically to handle global traffic.
* Should work well even on lower-end devices and slow internet connections.
* Users should have minimal lag,
* Strong encryption and data protection

***Fault Tolerance***

* If a microservice fails, it should be automatically restarted without affecting other services.
* Reduce chances of downtime.
* Failed network requests should be retried automatically.
* If a game server crashes, the user should be notified and redirected to a backup server.

***Technology and Design***

* Cloud platforms like AWS and Azure would be used to deploy the system worldwide.
* WebSocket-based services like Socket.IO for real-time updates across clients
* Each service runs in a container using Docker and is managed using Kubernetes which will help with load balancing, scaling, and failover.
* ZeroMQ for messaging between game servers.
* Redis or in-memory DB for game states
* A relational DB (PostgreSQL or SQLite) for user info, game histories, user levels and points and saved states.
* CDN provided by cloud services for traffic



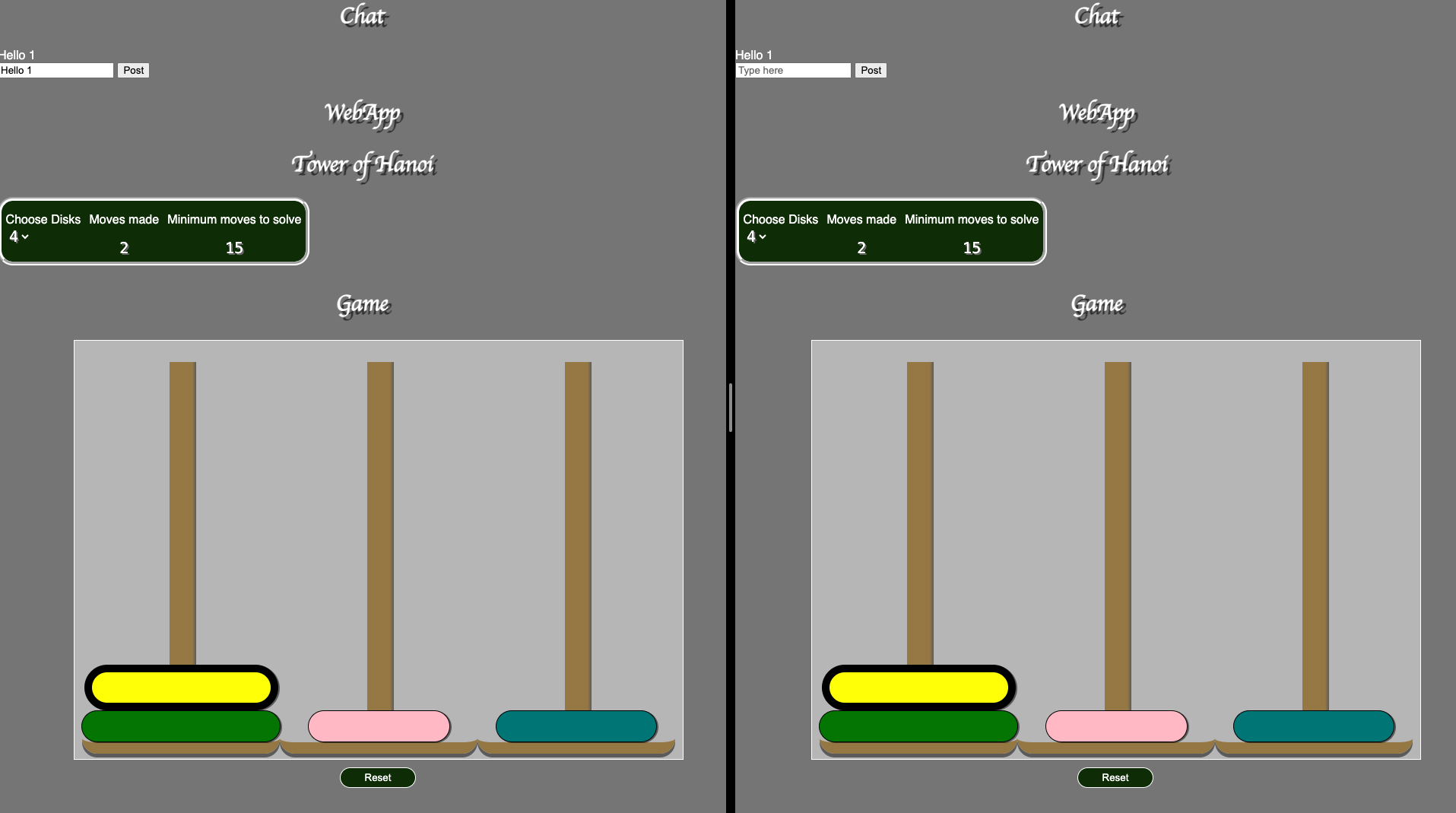
**2. Test Plan**

* Verify users can log in using credentials
* Test that users can create and join game rooms without errors.
* Ensure moves made by one player update on other players’ screens.
* Test that users who disconnect can rejoin games without losing progress.
* Check chat messages are sent during the game.
* Verify that the system retries or queues messages.
* Confirm that game progress is saved and can be restored after disconnects or page refreshes.
* check if other servers maintain game state without interruption.
* check if a new leader is elected automatically in case leader node failure

**Manual Testing - ZMQ**

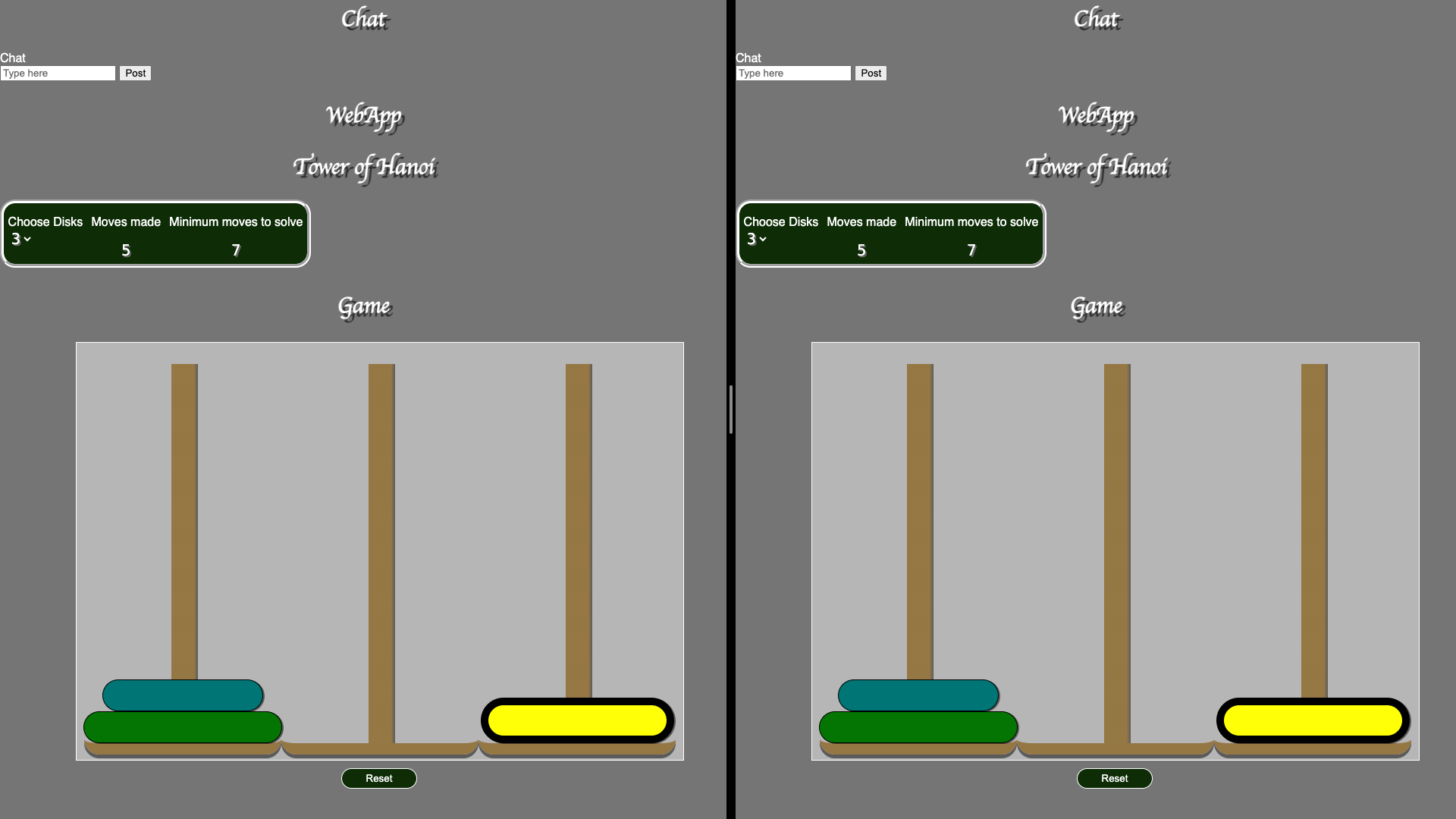
***Test 1 -*** *Basic Game Move Sync :* PASS

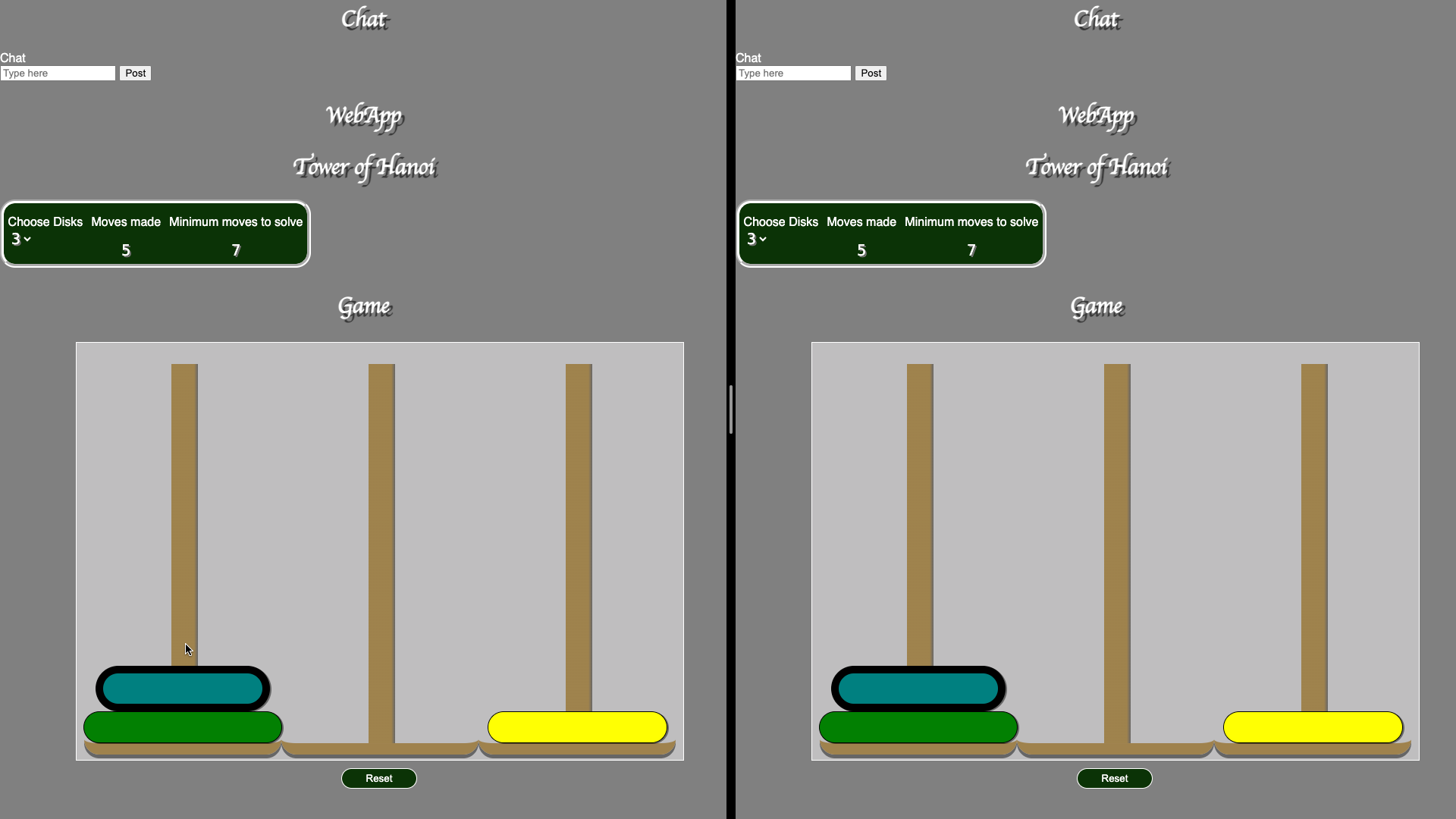
* Game state synced correctly after the move.



***Test 2 -*** *Invalid Move Rejected:* PASS

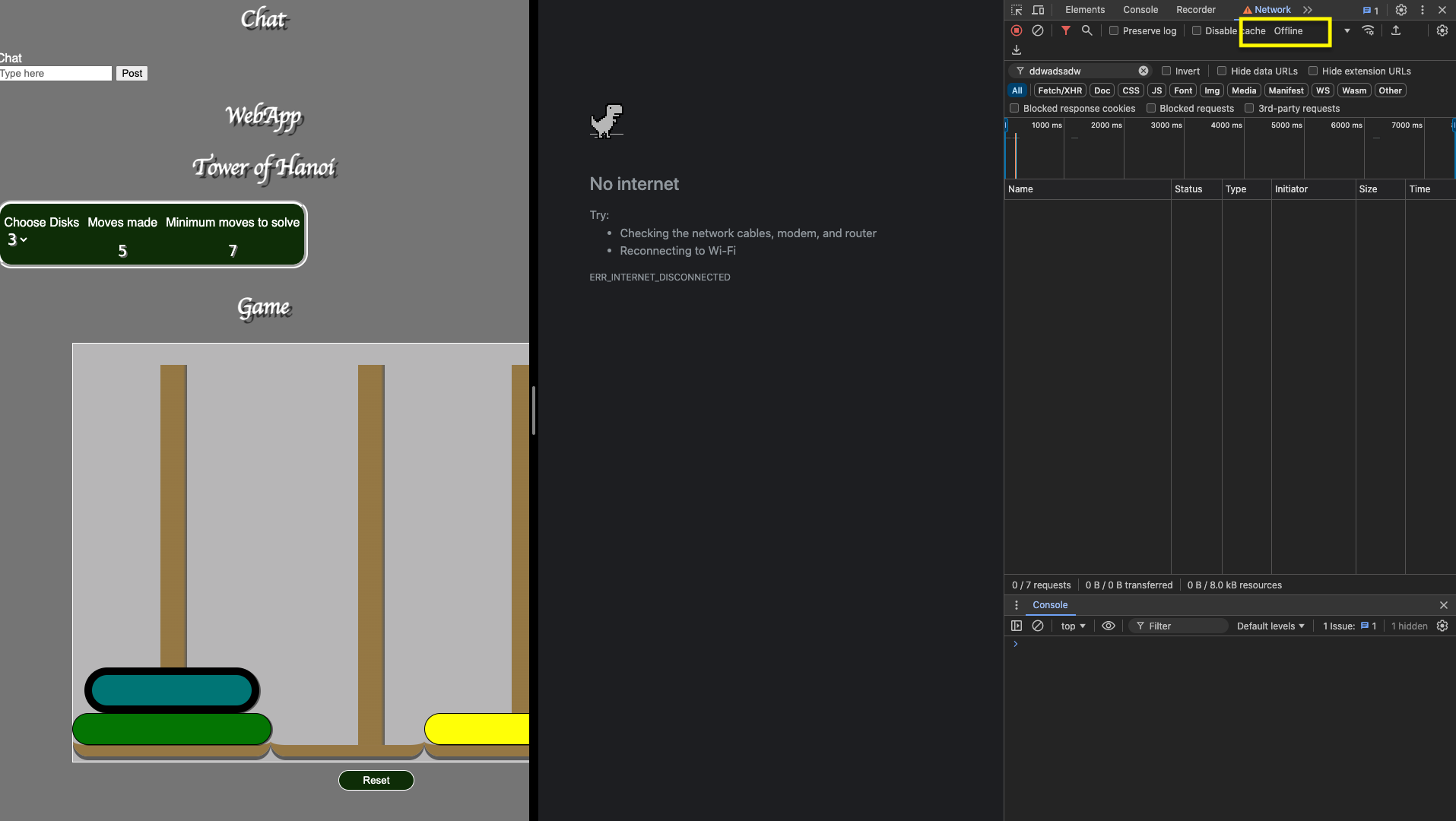
* Steps: Attempted to place a larger disk on a smaller one.
* Expected: Move should be blocked, and game state unchanged.

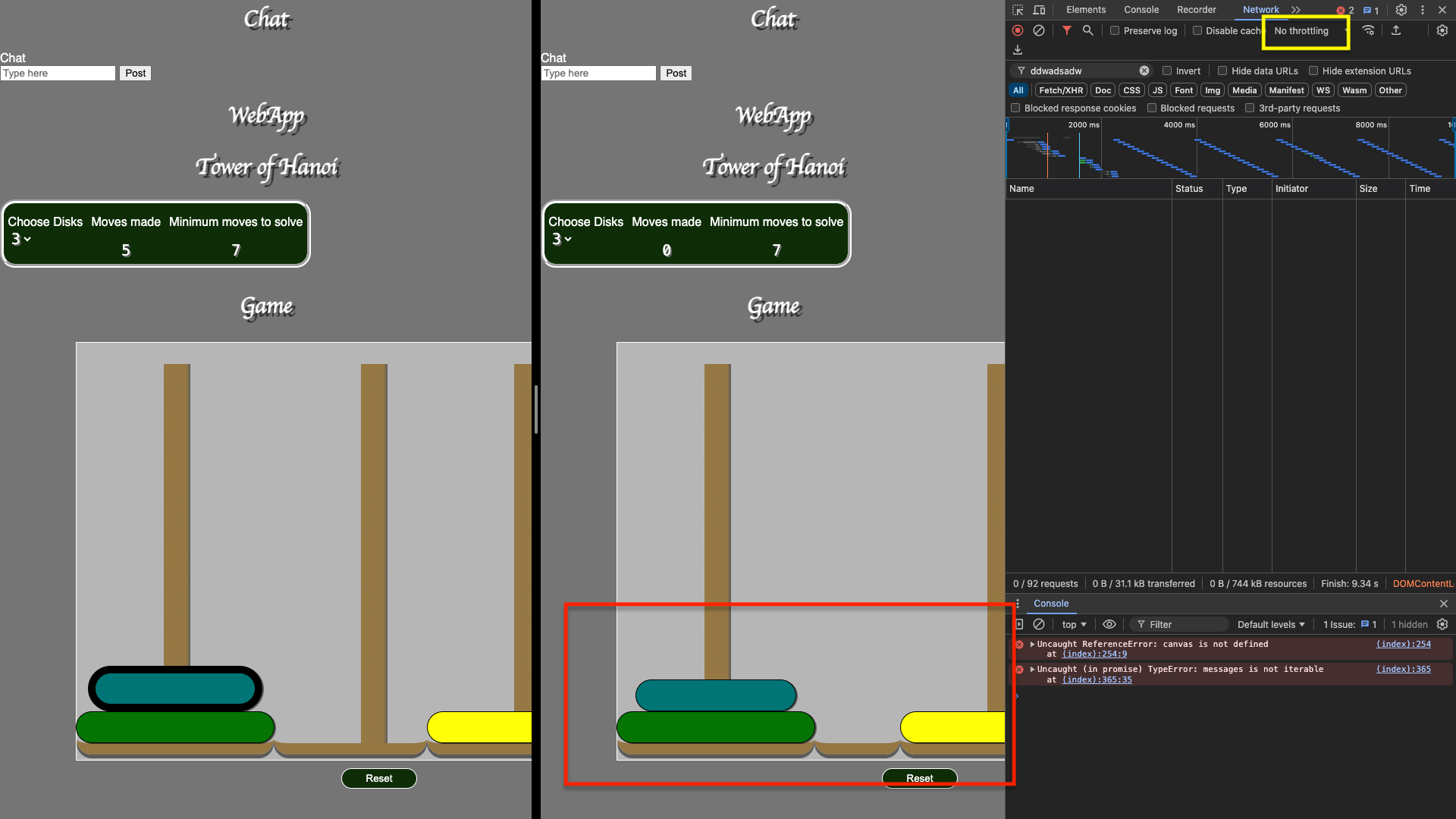
1. Click on a larger disk (yellow disk, trying to move on blue disk)
2. After clicking on tower 1, the yellow disk does not move. Instead, select the blue disk as a new move.



***Test 3*** *- Peer Crash:* PASS

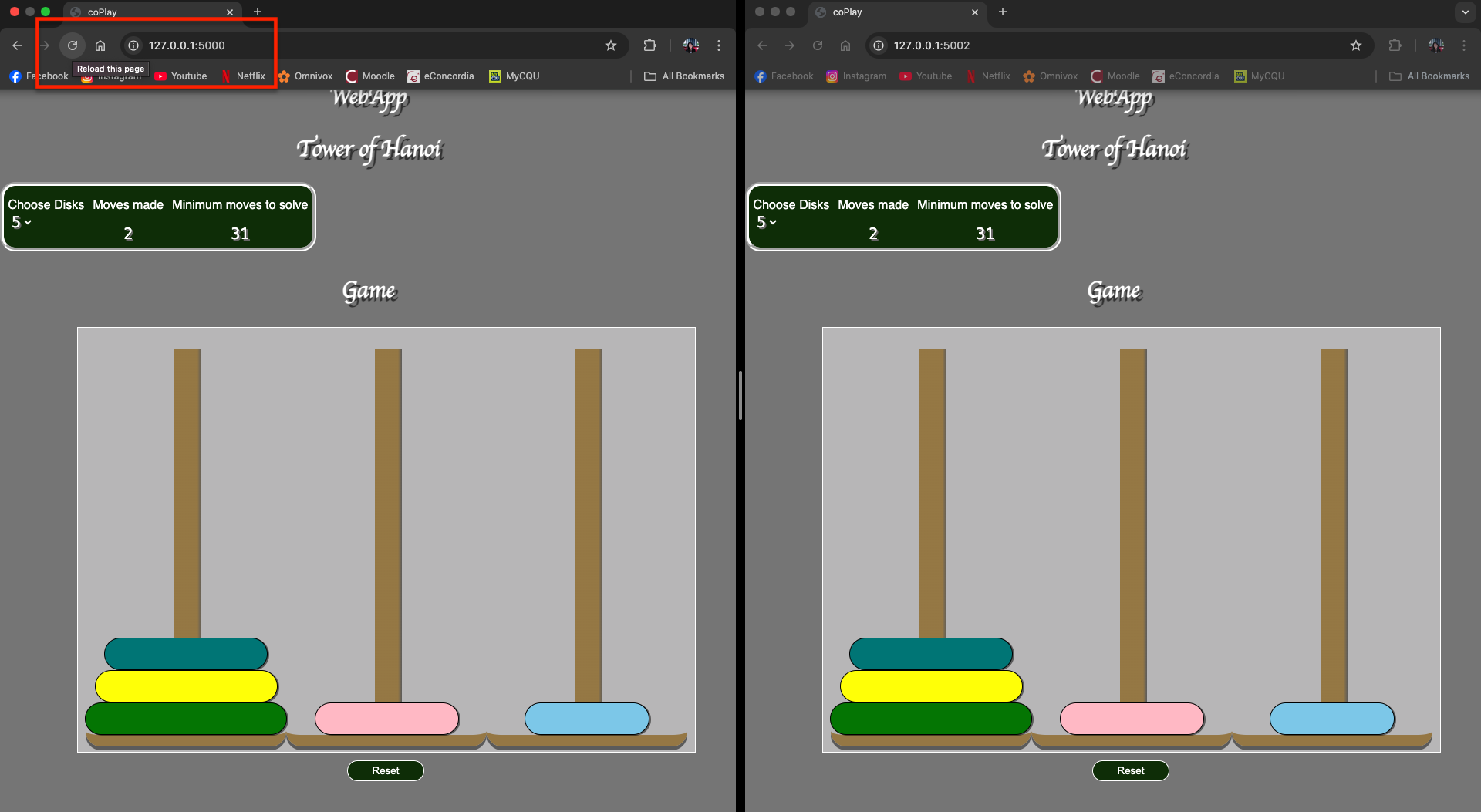
* Steps: Close a browser/server.
* Expected: On restart, the other user should reload the correct current state from others.

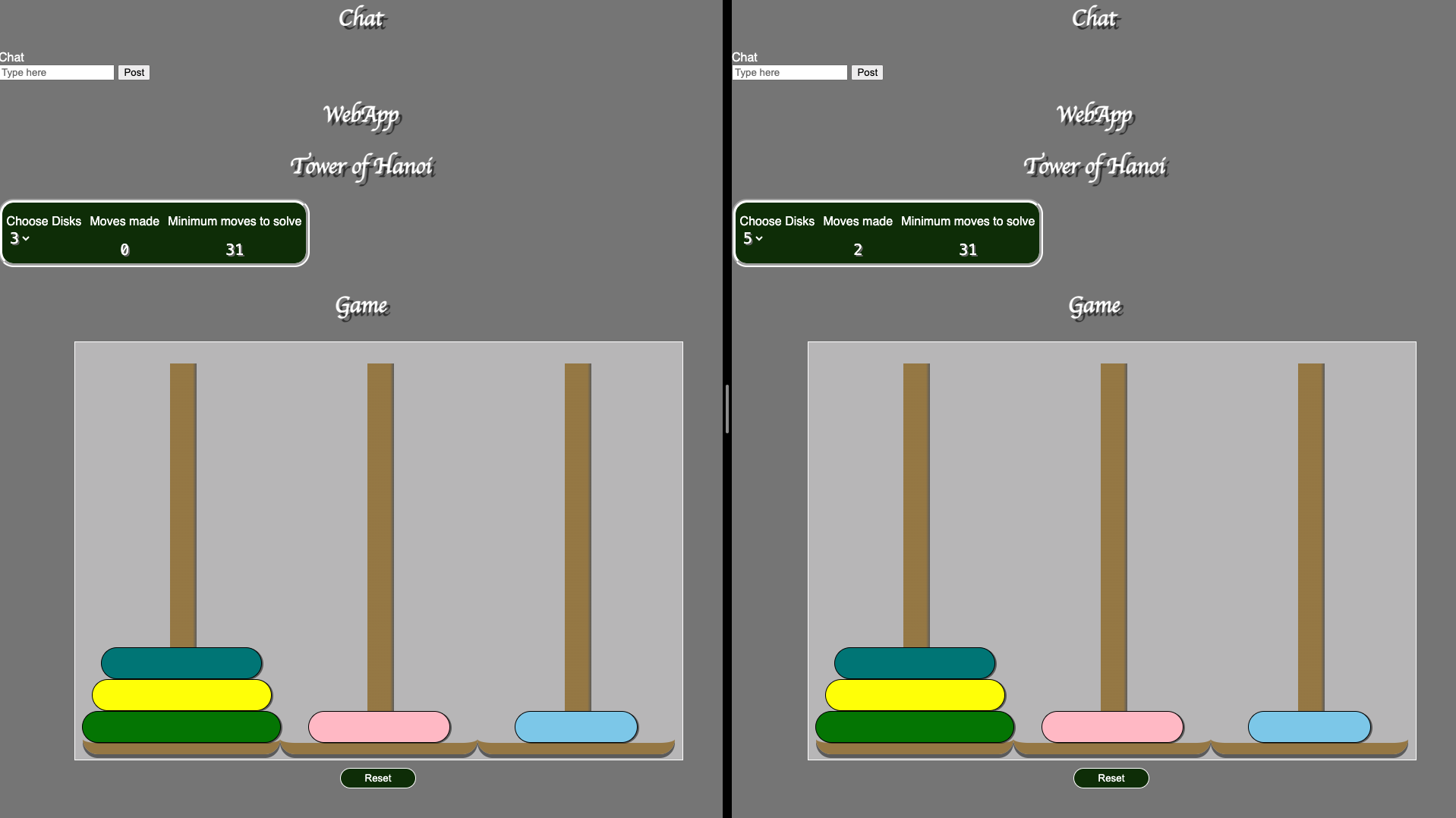
1. Open google dev tool.
2. Navigate to network tab, and select offline
3. Select “No throttling”
4. Page should refresh with current game state



***Test 4 - Page Refresh Sync:*** PASS

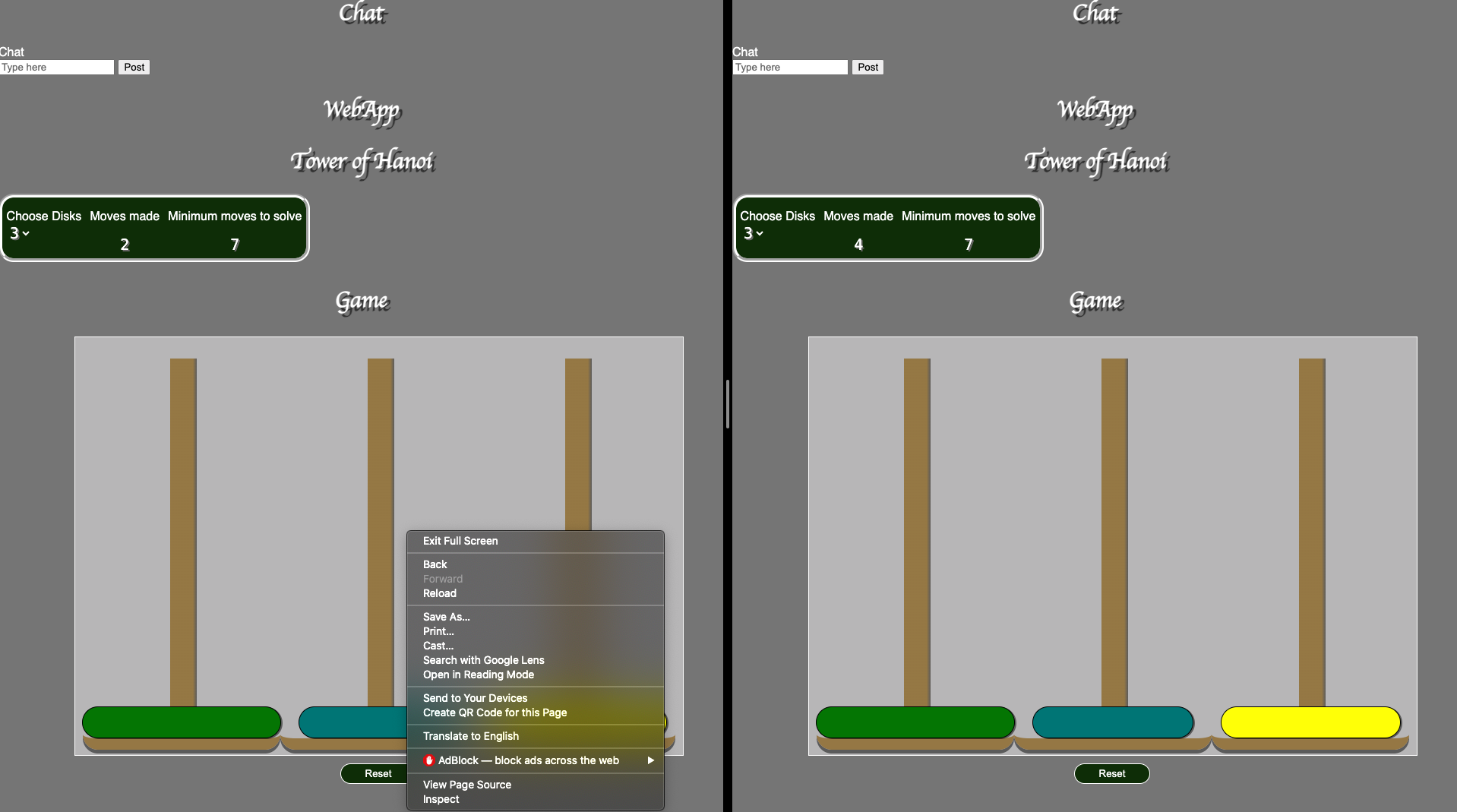
* Steps: Refreshed the browser tab on Client 1 after making a game move.
* Expected: Game and chat state should reload correctly from peers.

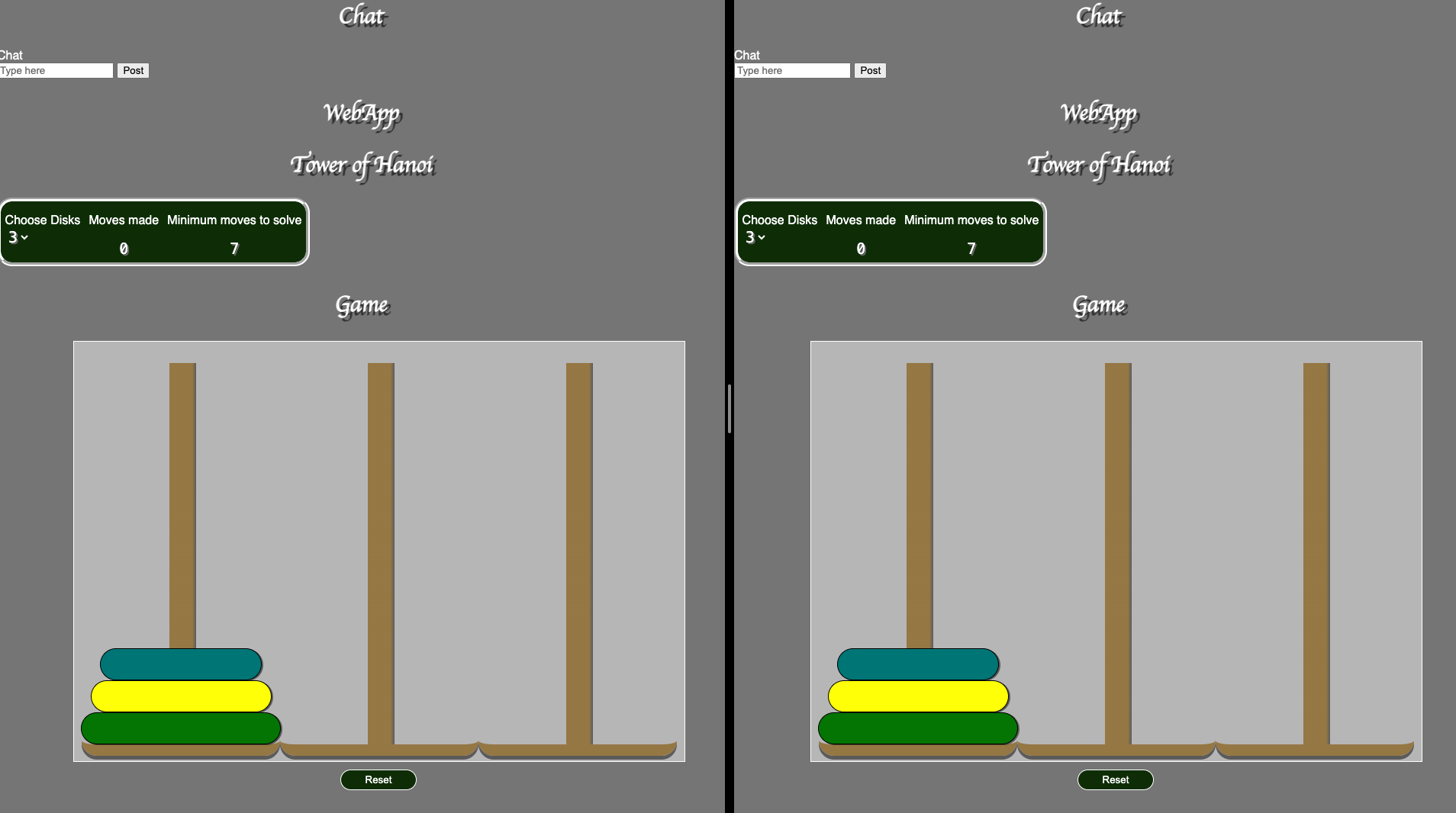
1. Make some moves on one endpoint
2. Ensure that both end reflect the game state
3. Reload the page
4. The page should load with the same game state



***Test 5 - New Game Reset Sync:*** PASS

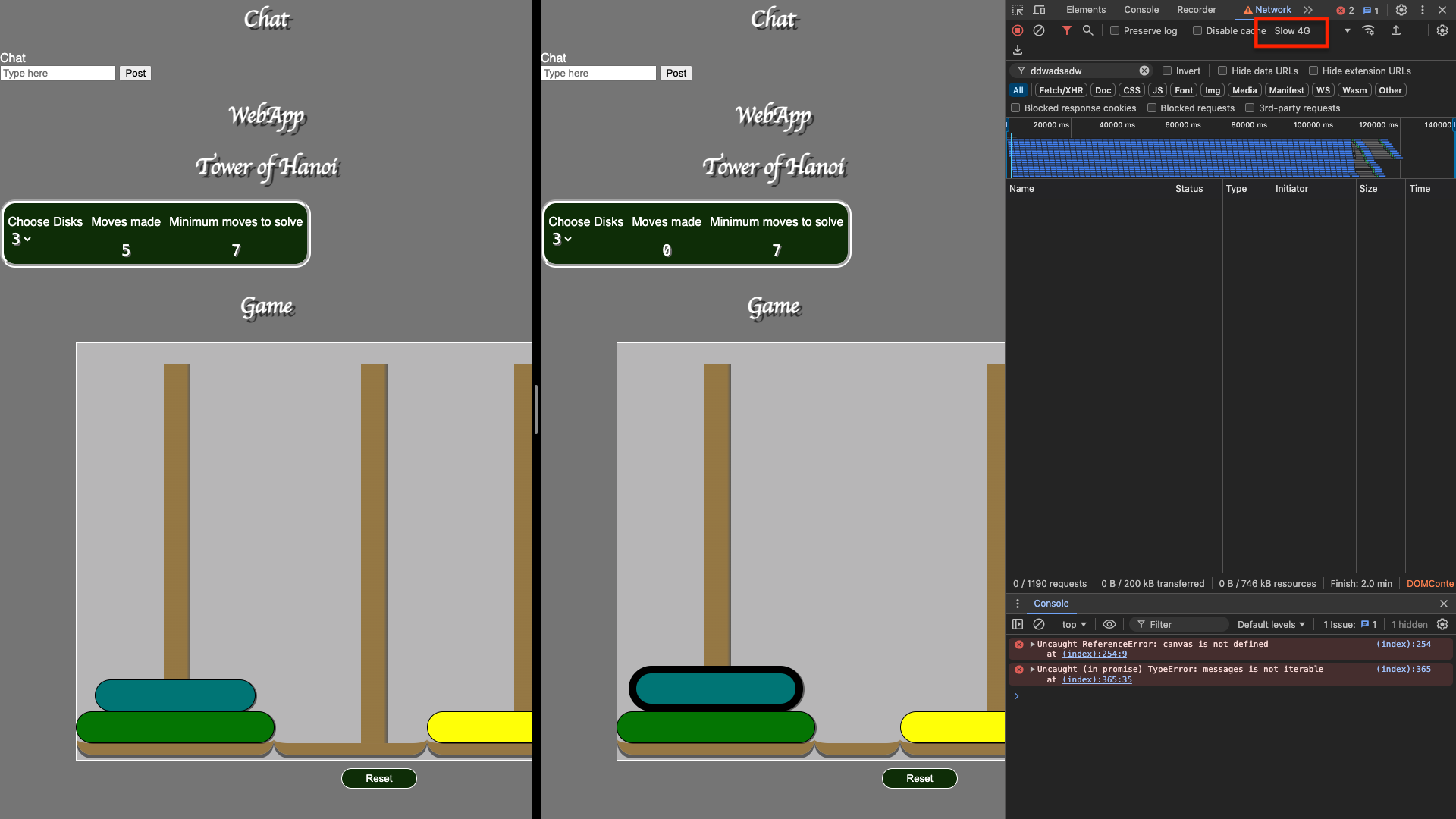
* Steps: Clicked "Reset" on one client.
* Expected: All clients should display the new game state.

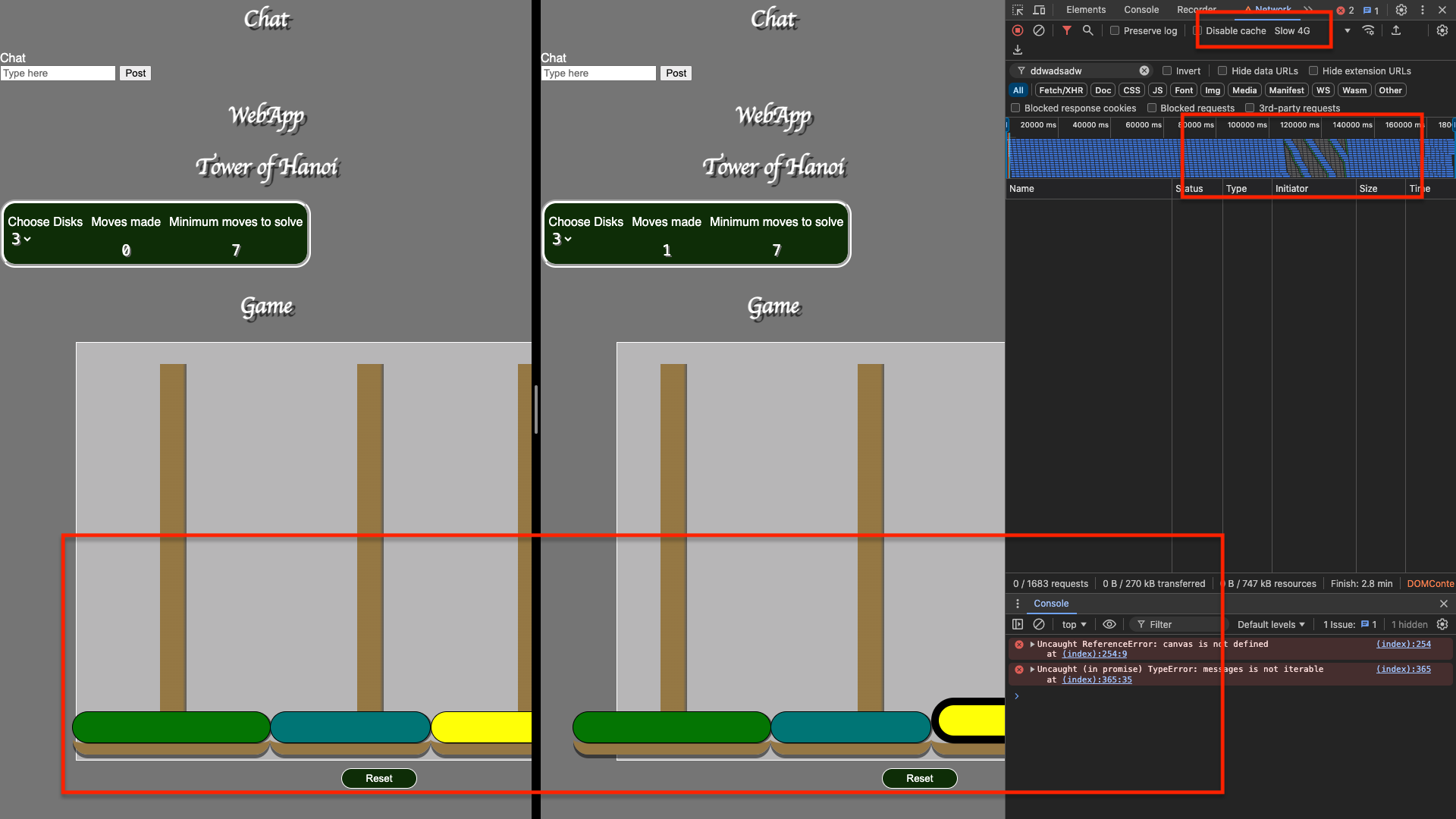
1. Make some moves.
2. Press the “reset” button on one of the client side
3. The game should restart for both clients.

******

***Test 6 - Network Lag:*** PASS

* Steps: Throttled network using Chrome DevTools
* Expected: Messages and game state should still sync correctly despite delay.

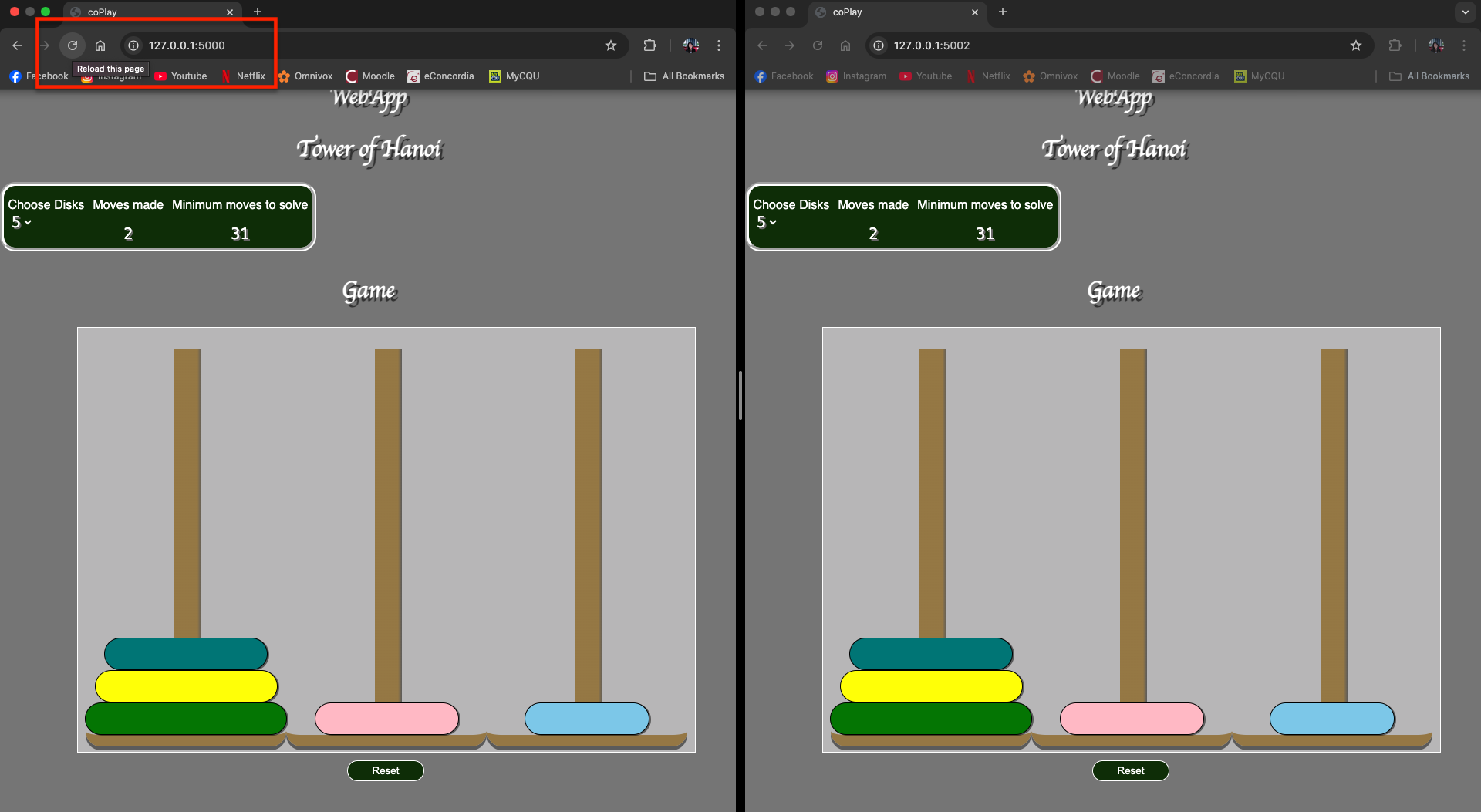
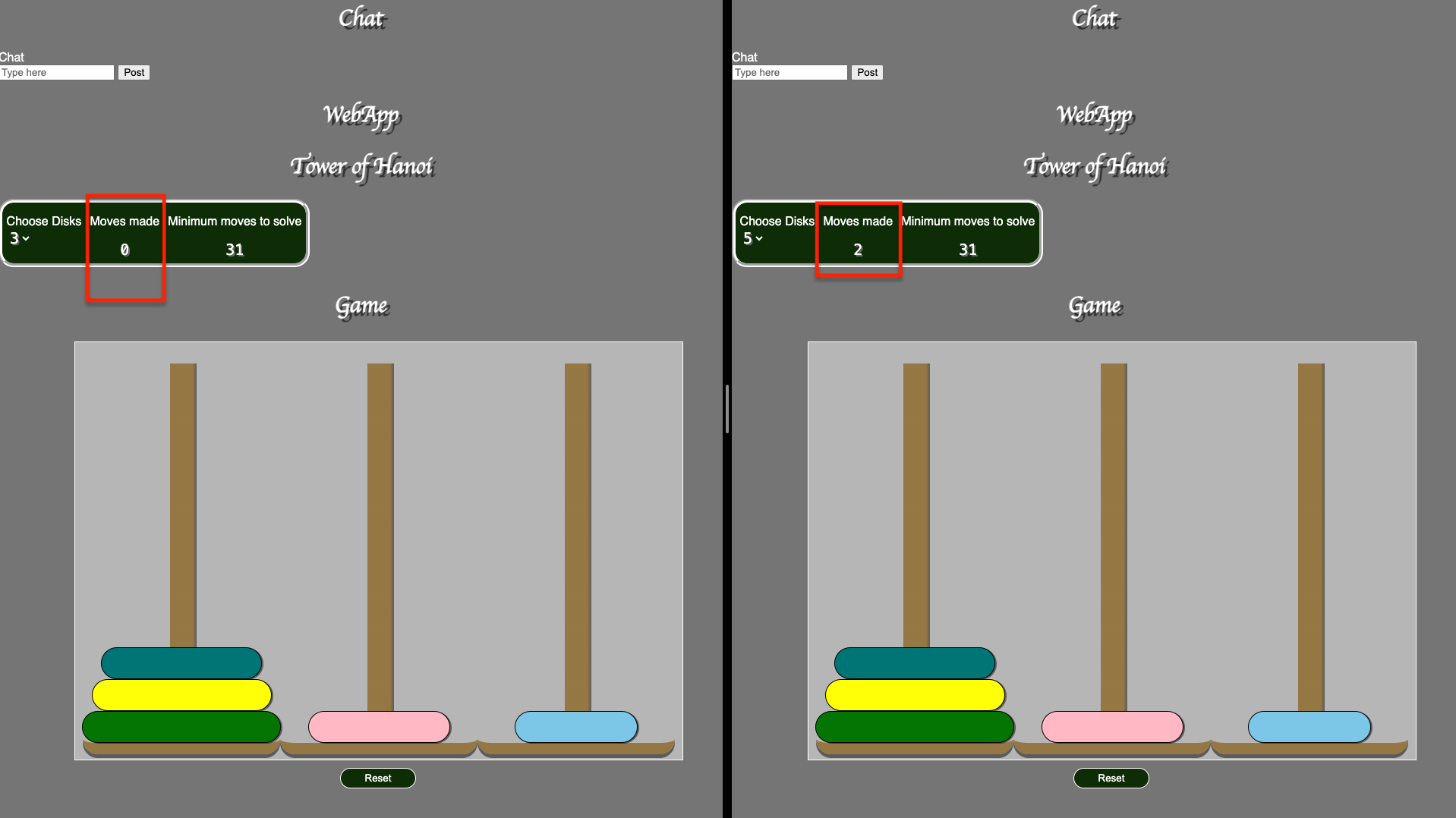
1. Open google dev tool.
2. Navigate to network tab, and select “Slow 4G”
3. Select a disk and move it to another tower



1. Despite the delay, the game state is the same on both endpoints

***Test 6 - Game move counter sync:*** FAIL

* Steps: Make a few moves, refresh page
* Expected: Move counter should restore previous number of moves me

1. Make some moves
2. Ensure that the counter increments
3. Reload the page
4. The counter should reset to “2”

[**https://www.geeksforgeeks.org/functional-vs-non-functional-requirements/#functional-requirements**](https://www.geeksforgeeks.org/functional-vs-non-functional-requirements/#functional-requirements)

[**https://www.paloaltonetworks.com.au/cyberpedia/kubernetes-docker**](https://www.paloaltonetworks.com.au/cyberpedia/kubernetes-docker)