Tetristributed: co-operative multiplayer tetris.

Harvard John A. Paulson School of Engineering and Applied Sciences

Dries P.C.M. Rooryck ‘26, Joseph D. Oronto-Pratt ‘26

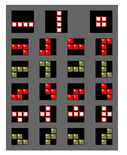
{dries_rooryck@college.harvard.edu, jorontopratt@college.harvard.edu}

Abstract & Motivation

Tetris is a universally recognized and deceptively simple game. We aimed to build a **co-op multiplayer version over the network** that we could deploy to production. The project forced us to tackle many distributed systems challenges, from the **client synchronization** issues of sharing real-time game state under network latency to the reliability issues of **2-fault-tolerance under server failures.**

We leverage the model of leader-follower server communication in a cluster with a **simple heartbeat election protocol** for failover, and use a methodical fine-grained board locking system to handle concurrency. We built our app on fullstack JavaScript, using React for frontend UI and Node for the webserver, and enhanced Socket.IO for cluster communication. Our app supports up to four players and concurrent games.

Game Mechanics



Aside from the synchronized multiplayer augmentations & enhancements, `Tetristributed` preserves much of the original core mechanics of classic Tetris that players are used to:

- **Rotation.** (Nintendo Rotation System from the original NES), movement. We opted for no wall-kicking in order to be faithful to this rotation system.
- **Drops.** Soft drops (holding the Down arrow key): 1 point per cell & hard drops (pressing Space Bar): 2 points per cell
- **Line clearing.** 100 points per line cleared.
- **Delayed auto shift.** Brief moment before left/right auto-shifting of pieces when Leff or Right arrow key is held.
- **Piece lock delay** after a customizable `LOCK_DELAY_FRAMES` to allow for sneaky move positioning and sliding while the piece is in contact with others.

Multiplayer enhancements:

- **Board size *increases*** with the number of players!
- Multiplayer **collision detection** to facilitate cooperation
- **All players view all scores** as they drop blocks and clear lines
- **Real-time** board & score updates shared to **all clients live**

Technical Challenges

- **Shared state consistency.** All players interact with a single shared board, so even minor desynchronization can break gameplay. We enforced server-authoritative updates on piece locks, line clears, and board state mutations to have all clients remain in sync. Deciding on a model of locking the board took many days.
- **UI synchronization with server ticks.** Even with smooth input, UI animations (lock flash, explosions) must reflect actual game logic and not client guesswork. We implemented server-controlled animation states broadcast at 60FPS to clients, with precise timers for line clear animations (30 frames), lock delays, and piece entry timing so all players see the same visual effects synchronized with the authoritative game state.
- **Server failover.** Gameplay must persist even in the face of server failure and player disconnection, so we implemented leader-follower replication, plus replication of fine-grained game state and player state, with a ‘rejoinRoom’ event allowing players to continue gameplay. This took over 30 hours.
- **Line-clear explosion animations:** Rendering dynamic particle explosions as the line clear animation brought on performance challenges, especially when hundreds of particles need to move smoothly or when a player clears several lines at once. We thus optimized animations using React’s `useMemo` and `requestAnimationFrame` to keep fluid 60 FPS gameplay without overwhelming the browser rendering pipeline.

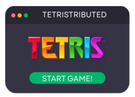
Cluster Design

- Three server replicas run on unique ports (3001, 3002, 3003) and maintain inter-server communication through dedicated cluster ports (4001, 4002, 4003).
- The `ClusterManager` class coordinates this distributed system, implementing heartbeat monitoring, leader election, and state replication across the cluster. When the leader fails, the remaining servers detect this through missed heartbeats and initiate a leader election process based on server IDs, with the lowest available ID becoming the new leader.
- Client connections managed by `ServerConnectionManager`, which detects the leader and connects to it.
- All in all, this allows for **mid-game server failure and multiple client reconnection to another server to keep playing without skipping a beat, just by reloading the browser.**

Server Design

- The game logic centers around room-based multiplayer sessions (plural!), with rooms identified by unique codes.
- Each room maintains its own game state, including the tetromino board, active players, and scoring.
- The server implements classic Tetris mechanics like piece locking, line clearing, and collision detection, but extends this for multiplayer interactions. The server features sophisticated game state management with proper timing mechanisms (using setInterval for game loops running at 60 FPS), delayed auto-shift for smooth controls, and elegant handling of piece rotation.
- A **centralized server design** was chosen given that in co-op tetris, real-time synchronization becomes a big issue, and a P2P network would make it much harder to maintain an authoritative single-source of truth about game state.

Frontend Sequencing & UI



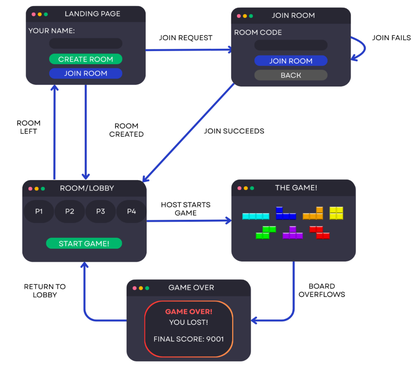
The frontend is a React application built to guide players through the game with a clear flow:

- **Home screen.** The entry point, with name selection, room creation & joining functionality, and not much else.
- **Lobby.** Pre-game room with a player list. The host can start the game with players who have readied up, and not before that! Players can join by entering the right 6 digit room-code,
- **Gameplay.** Each player controls their own Tetris piece on the same Tetris board, rendered using HTML5 canvas and React Konva. Block movement follows a request-response model with the server so that clients agree on block position, as do other authoritative actions (spawning, piece locking, line clears).
- **Game Over Screen.** Gameplay continues indefinitely until the board is overflowed. Afterwards, players are presented with a game over screen and then returned to the lobby to spin up another game!

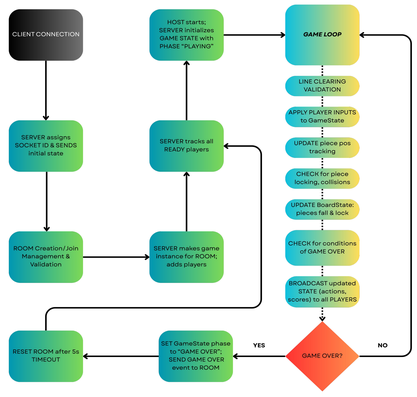
The UI was deliberately kept simple and snappy:

- We wanted to keep the retro arcade feel of Tetris, opting for buttons and avoiding pop-ups.
- Some features include **smooth animations** with canvas rendering (e.g. pixel explosions upon line clearing), real-time UI updates via **socket.io event listeners**, timers, rotating background images, being inspired by design elements of Tetris NPM 2.

Frontend Sequencing Flow



Server Flow



Testing

Tetristributed's testing pipeline involved:

- **Stress and load testing.** Done by us and other members of the SEAS community in order to stress-test the game and hunt down bugs. Simulating many clients connecting to the server.
- **Fault injection.** We intentionally triggered client disconnections and server restarts to validate our replication, rejoining, and fault-tolerance.
- **Game logic testing:** pretty self-evident.
- **Debug logging.** We toggle detailed server-side and client-side logs to trace socket events, game state transitions, room creation, and player actions.
- **Unit testing:** we have a framework to test each function.
- **Integration testing:** we test client-server communication for each event type.

Codebase

Future Work

Deployment: Buying a web-domain and hosting our fun game on the internet via Vercel.
Scaling: adding matchmaking systems for a large userbase, distributing handling different instances of the games to different regional servers. This would relieve the bottleneck on the number of clients connected..
Move sequencing: Trying a different method of sequencing moves using a log and Lamport Clocks.
Decentralization: As an exercise, try our hand at a decentralized version, where clients communicate peer-to-peer.
Security improvements: detecting client tampering & cheating attempts using proof-of action.
Game mechanics improvements: implement piece-drop speed-up, tournament, persistent store of game records. Failover improvements: True Paxos, nginx for load-balancing and automatic reconnection rather than ctrl+r.

Acknowledgements

We thank **Jim Waldo** and the Spring 2025 CS 2620 staff. In particular, we thank Sibi and Swati, our GOATs. Joseph contributed to the front-end sequencing, animation, UI, and design, while Dries focused on server failover and game state handling. Our code can be found at:

https://github.com/drooryck/distributed-systems/tree/main/tetris.