Read the instructions in this guide in their entirety before beginning the project.

Creating a load-balancing server for handling multiple game instances will improve scalability and ensure smooth performance as the number of players grows. In this guide, we’ll cover setting up a load balancer with Node.js and Express, using multiple game instances, and directing player connections to different servers. This approach distributes the player load evenly across game instances, enhancing game responsiveness.

**Project Overview**

The goal is to build a load-balancing system for the game, which distributes players across multiple game servers. Each game server runs its own instance of the game using Socket.IO. A load balancer will manage connections, distributing them evenly to ensure each server handles an optimal number of players.

**1. Setting Up the Infrastructure**

**Project Structure**

Your load-balancing project should have the following structure:

bash

Copy code

/load-balancer-project

│

├── /public # Frontend files

│ ├── index.html # Main HTML page

│ ├── style.css # Optional CSS

│ ├── game.js # Game logic (client-side)

│

├── /game-server # Folder for individual game servers

│ └── server.js # Game server code with Socket.IO

│

├── loadBalancer.js # Main load balancer server

├── package.json # Project dependencies

└── README.md # Project documentation

**Install Dependencies**

You’ll need several Node.js packages to set up your load balancer and game servers:

* **Express**: For serving files and routing HTTP requests.
* **Socket.IO**: For WebSocket communication between the game servers and clients.
* **HTTP-Proxy**: To create a reverse proxy for forwarding requests from the load balancer to individual game servers.

Install the dependencies by running:

bash

Copy code

npm install express socket.io http-proxy

**2. Creating Game Server Instances**

Each game server instance is essentially a separate Node.js application with its own instance of Express and Socket.IO. Every game server instance manages players assigned to it, updates the game state, and broadcasts relevant data to connected clients.

**Game Server Setup**

In the server.js file within the /game-server directory:

1. **Create the Express and Socket.IO Server**: Set up an Express server to serve game files and listen for connections. Attach Socket.IO to handle real-time communication with clients.
2. **Game Instance Management**: Handle player connections, maintain a list of connected players, and manage game updates (like player positions, scores, etc.). Each instance should only handle the players assigned by the load balancer.
3. **Port Management**: Each game server runs on a unique port (e.g., 3001, 3002, etc.), allowing the load balancer to route traffic accordingly.

Run multiple game servers on different ports to prepare for load balancing. For testing, you can start multiple game instances manually or with a process manager like PM2 to keep them running.

**3. Setting Up the Load Balancer**

**Load Balancer Responsibilities**

The load balancer serves as the entry point for all incoming player connections. Its primary responsibilities include:

* **Distributing Connections**: Direct new player connections to the game server with the lowest load.
* **Balancing Load**: Track the number of active connections per server to balance the load.
* **Monitoring Health**: Check that each game server is functioning properly. If a server goes down, the load balancer should reassign its players.

**Implementing the Load Balancer**

In the loadBalancer.js file:

1. **Set Up Express and HTTP-Proxy**: Use Express for the main load-balancer server and http-proxy for routing requests to game servers.
2. **Track Server Load**: Maintain a record of the number of active connections per game server. This will help in balancing the load effectively.
3. **Redirect Connections**: When a new player connects, the load balancer forwards the connection to the server with the lowest load. HTTP-Proxy can redirect WebSocket connections as well, ensuring real-time communication is maintained.

**4. Managing Player Connections Across Game Servers**

**Player Assignment**

When a player connects to the load balancer:

1. **Identify the Least Loaded Server**: Determine which game server has the fewest active players.
2. **Forward the Connection**: Direct the player’s connection to the identified server instance.
3. **Update Load Tracking**: Increment the load count for that server to keep track of active connections.

**Server Failover**

If a game server becomes unreachable, the load balancer should:

* **Remove the Server from Available List**: Temporarily remove the server from the pool.
* **Redistribute Players**: Reassign any new connections to other available servers.
* **Retry**: Regularly check to see if the failed server has recovered and can be added back to the pool.

**5. Synchronizing Game State Across Servers**

For a game that requires players to interact with each other in real time, you might need a central source of truth for game state. Here’s an approach to keep states synchronized without overwhelming the servers:

1. **Shared State Storage (Optional)**: Use an external database (e.g., Redis) to store and retrieve game states in real time. Game servers can read/write to this central database to ensure data consistency.
2. **Inter-Server Communication**: Use Socket.IO's rooms feature or a message broker like Redis Pub/Sub to let game servers communicate and broadcast events to players on other servers when necessary.

If each game server instance only manages players within its own game, you may not need full state synchronization across servers, simplifying the setup.

**6. Frontend Connection to Load Balancer**

In game.js, set up the Socket.IO client to connect to the load balancer rather than directly to a game server. This way, each player will be automatically directed to an available game server without requiring a manual selection.

1. **Initial Connection**: The frontend connects to the load balancer, which handles server assignment.
2. **Dynamic Assignment**: Based on load, the player will be routed to a game server.
3. **Stable Connection**: The load balancer manages player connections continuously, allowing players to join a less crowded server for an optimal experience.

**7. Reading Instructions**

Immediately stop reading this paper. Do not say anything to your classmates about this step. Go to this week’s DocPac and write all of the bulleted items in the “Events” portion of the DocPac in the blank box on the back of the DocPac. Do not complete ay of the other steps in this guide.

**8. Testing and Scaling**

**Testing Load Distribution**

Test the setup by connecting multiple players and observing how connections are distributed across game servers. You should see players being directed to different servers based on their load, with each server’s load count updated in real time.

**Scaling**

For larger applications, consider using cloud-based solutions (like AWS Load Balancer or NGINX as a reverse proxy) for more robust load distribution. You can also deploy additional game servers to handle increased player loads.

**9. Optimization and Fail-Safety**

Auto-Scaling: Use automated scaling tools or scripts to dynamically launch new game instances during peak times and reduce instances when traffic decreases. Cloud platforms like AWS or Google Cloud can manage this scaling automatically.

Improving Latency: Minimize latency between game servers and clients by optimizing server selection or using a geographic distribution approach, connecting players to the nearest server.

Graceful Error Handling: Design the frontend to notify players if the **load balancer is unable to assign a server, ensuring a smooth user experience.**