```
const express = require('express');
const http = require('http');
const socketIO = require('socket.io');
const path = require('path');
const { PlayerPresenceManager, PlayerState } = require('./public/js/playerPresenc

const app = express();
const server = http.createServer(app);
const io = socketIO(server, {
    cors: {
        origin: "*",
        methods: ["GET", "POST"]
    }
});

const PORT = process.env.PORT || 3000;
const MAX_DRAW_ATTEMPTS = 50; // Safety limit for "draw until match" to prevent i

/* ======================
   STATIC FILES & ROUTES
====================== */

app.use(express.static(path.join(__dirname, 'public')));

app.get('/', (req, res) => {
    res.sendFile(path.join(__dirname, 'public', 'index.html'));
});

app.get('/health', (req, res) => {
    res.json({
        status: 'ok',
        rooms: rooms.size,
        lobbies: Object.keys(lobbies).length
    });
});

app.get('*', (req, res) => {
    if (!req.url.includes('.')) {
        res.sendFile(path.join(__dirname, 'public', 'index.html'));
    } else {
        res.status(404).send('File not found');
    }
});
```

```
/* ======================
   GAME STORAGE
====================== */

const rooms = new Map();
const lobbies = {};
const lobbyPresenceManagers = new Map(); // lobbyId -> PlayerPresenceManager

/* ======================
   GAME ROOM CLASS
====================== */

class GameRoom {
    constructor(roomId, players, settings) {
        this.roomId = roomId;
        this.players = players.map(p => ({
            id: p.id,
            name: p.name,
            hand: [],
            calledUno: false
        }));
        this.deck = [];
        this.discardPile = [];
        this.currentPlayer = 0;
        this.currentColor = null;
        this.currentValue = null;
        this.direction = 1;
        this.stackedDrawCount = 0;
        this.stackStartedBy = null;
        this.hasDrawnThisTurn = false;
        this.settings = settings || {};
        this.gameStarted = false;
    }

    createDeck() {
        const COLORS = ['red', 'blue', 'green', 'yellow'];
        const NUMBERS = ['0','1','2','3','4','5','6','7','8','9'];
        const ACTIONS = ['Skip', 'Reverse', '+2'];

        this.deck = [];

        COLORS.forEach(color => {
            this.deck.push({ color, value: '0', type: 'number' });

            for (let i = 0; i < 2; i++) {
                NUMBERS.slice(1).forEach(n =>
                    this.deck.push({ color, value: n, type: 'number' })
```

```
            );
            ACTIONS.forEach(a =>
                this.deck.push({ color, value: a, type: 'action' })
            );
        }
    });

    for (let i = 0; i < 4; i++) {
        this.deck.push({ color: 'wild', value: 'Wild', type: 'wild' });
        this.deck.push({ color: 'wild', value: 'Wild+4', type: 'wild' });
    }

    this.shuffleDeck();
}

shuffleDeck() {
    for (let i = this.deck.length - 1; i > 0; i--) {
        const j = Math.floor(Math.random() * (i + 1));
        [this.deck[i], this.deck[j]] = [this.deck[j], this.deck[i]];
    }
}

dealCards(count = 7) {
    this.players.forEach(p => {
        p.hand = [];
        for (let i = 0; i < count; i++) {
            p.hand.push(this.deck.pop());
        }
    });

    let start;
    do {
        start = this.deck.pop();
    } while (start.type !== 'number');

    this.discardPile = [start];
    this.currentColor = start.color;
    this.currentValue = start.value;
}

getGameState(playerId) {
    const index = this.players.findIndex(p => p.id === playerId);
    const currentPlayerIndex = this.currentPlayer;
    const currentPlayerName = this.players[currentPlayerIndex]?.name || 'Unkn

    return {
        roomId: this.roomId,
```

```
        yourHand: this.players[index].hand,
        yourName: this.players[index].name,
        opponents: this.players.filter((_, i) => i !== index).map(p => ({
            id: p.id,
            name: p.name,
            cardCount: p.hand.length
        })),
        currentPlayer: this.currentPlayer,
        currentPlayerName: currentPlayerName,
        isYourTurn: index === this.currentPlayer,
        discardPile: this.discardPile.at(-1),
        currentColor: this.currentColor,
        currentValue: this.currentValue,
        deckCount: this.deck.length,
        settings: this.settings,
        stackedDrawCount: this.stackedDrawCount,
        stackStartedBy: this.stackStartedBy,
        hasDrawnThisTurn: this.hasDrawnThisTurn
    };
}

canPlayCard(card) {
    if (!card) return false;
    if (card.type === 'wild') return true;

    if (this.settings.allowStacking && this.stackedDrawCount > 0) {
        if (this.currentValue === '+2' && card.value === '+2') return true;
        if (this.currentValue === 'Wild+4' && card.value === 'Wild+4') return
        return false;
    }

    return card.color === this.currentColor || card.value === this.currentVal
}

playCard(playerId, cardIndex, chosenColor) {
    const playerIndex = this.players.findIndex(p => p.id === playerId);
    if (playerIndex === -1) {
        return { success: false, error: 'Player not found' };
    }

    const player = this.players[playerIndex];
    const card = player.hand[cardIndex];

    if (!card) {
        return { success: false, error: 'Invalid card' };
    }
```

```
// Check for jump-in: Allow play if not current player but jump-in is ena
// and card is an exact match (same color AND value, number cards only)
if (playerIndex !== this.currentPlayer) {
    const isJumpIn = this.settings.allowJumpIn &&
                  card.type === 'number' &&
                  card.color === this.currentColor &&
                  card.value === this.currentValue;

    if (!isJumpIn) {
        return { success: false, error: 'Not your turn' };
    }

    // Valid jump-in: Set current player to this player
    this.currentPlayer = playerIndex;
}

if (!this.canPlayCard(card)) {
    return { success: false, error: "Can't play that card" };
}

// Remove card from player's hand
player.hand.splice(cardIndex, 1);
this.discardPile.push(card);

// Handle wild cards
if (card.type === 'wild') {
    this.currentColor = chosenColor || 'red';
    this.currentValue = card.value;

    if (card.value === 'Wild+4') {
        if (this.settings.allowStacking) {
            if (this.stackedDrawCount === 0) {
                this.stackStartedBy = this.players[playerIndex].name;
            }
            this.stackedDrawCount += 4;
        }
        // Non-stacking draw is handled in handleCardEffect
    }
} else {
    this.currentColor = card.color;
    this.currentValue = card.value;
}

// Handle special cards
this.handleCardEffect(card, playerIndex);

// Check for O,no penalty after playing card
```

```
        const unoCheck = this.checkUnoAfterPlay(playerIndex);

        // Build draw animation payload for +2/+4 cards
        // - Non-stacking: cards land on victim immediately
        // - Stacking: show the card being added to the pending stack (victim TBD
        let drawAnimation = null;
        const nextIdx = this.getNextPlayerIndex(playerIndex);
        if (card.value === '+2') {
            drawAnimation = {
                victimId: this.players[nextIdx]?.id,
                victimName: this.players[nextIdx]?.name,
                playerId: this.players[playerIndex].id,
                playerName: this.players[playerIndex].name,
                count: 2,
                cardValue: '+2',
                stacking: !!this.settings.allowStacking
            };
        } else if (card.value === 'Wild+4') {
            drawAnimation = {
                victimId: this.players[nextIdx]?.id,
                victimName: this.players[nextIdx]?.name,
                playerId: this.players[playerIndex].id,
                playerName: this.players[playerIndex].name,
                count: 4,
                cardValue: 'Wild+4',
                stacking: !!this.settings.allowStacking
            };
        }

        return {
            success: true,
            winner: player.hand.length === 0 ? playerIndex : null,
            unoPenalty: unoCheck.penaltyApplied ? { playerName: unoCheck.playerNa
            drawAnimation
        };
    }

    handleCardEffect(card, playerIndex) {
        switch(card.value) {
            case 'Skip':
                this.skipNextPlayer();
                break;

            case 'Reverse':
                this.direction *= -1;
                if (this.players.length === 2) {
                    // In 2-player, Reverse acts like Skip
```

```
            this.skipNextPlayer();
        } else {
            this.advanceTurn();
        }
        break;

    case '+2':
        if (this.settings.allowStacking) {
            if (this.stackedDrawCount === 0) {
                this.stackStartedBy = this.players[playerIndex].name;
            }
            this.stackedDrawCount += 2;
            this.advanceTurn();
        } else {
            this.drawCards(this.getNextPlayerIndex(), 2);
            this.skipNextPlayer();
        }
        break;

    case 'Wild+4':
        if (this.settings.allowStacking) {
            if (this.stackedDrawCount === 0) {
                this.stackStartedBy = this.players[playerIndex].name;
            }
            this.stackedDrawCount += 4;
            this.advanceTurn();
        } else {
            this.drawCards(this.getNextPlayerIndex(), 4);
            this.skipNextPlayer();
        }
        break;

    case '0':
    case '7':
        if (this.settings.allowSpecial07) {
            this.swapHands(playerIndex);
        }
        this.advanceTurn();
        break;

    case '4':
        if (this.settings.allowSpecial48) {
            // 4 skips next player
            this.skipNextPlayer();
        } else {
            this.advanceTurn();
        }
```

```
            break;

        case '8':
            if (this.settings.allowSpecial48) {
                // 8 reverses direction
                this.direction *= -1;
                if (this.players.length === 2) {
                    // In 2-player, Reverse acts like Skip
                    this.skipNextPlayer();
                } else {
                    this.advanceTurn();
                }
            } else {
                this.advanceTurn();
            }
            break;

        default:
            // Regular card or wild
            this.advanceTurn();
    }
}

skipNextPlayer() {
    // Advance turn twice to skip the next player
    this.advanceTurn();
    this.advanceTurn();
}

swapHands(playerIndex) {
    // Swap hands with next player (respecting direction)
    const nextPlayerIndex = this.getNextPlayerIndex(playerIndex);
    const temp = this.players[playerIndex].hand;
    this.players[playerIndex].hand = this.players[nextPlayerIndex].hand;
    this.players[nextPlayerIndex].hand = temp;
}

getNextPlayerIndex(fromIndex = null) {
    const index = fromIndex !== null ? fromIndex : this.currentPlayer;
    return (index + this.direction + this.players.length) % this.players.leng
}

advanceTurn() {
    this.currentPlayer = this.getNextPlayerIndex();
    this.hasDrawnThisTurn = false; // reset draw flag for the new player's tu

    // Handle stacked draws
```

```
    if (this.stackedDrawCount > 0) {
        const stackableCard = this.players[this.currentPlayer].hand.find(card
            (this.currentValue === '+2' && card.value === '+2') ||
            (this.currentValue === 'Wild+4' && card.value === 'Wild+4')
        );

        if (!stackableCard) {
            // Player can't stack, must draw
            this.drawCards(this.currentPlayer, this.stackedDrawCount);
            this.stackedDrawCount = 0;
            this.stackStartedBy = null;
            this.advanceTurn();
        }
    }
}

drawCards(playerIndex, count) {
    const player = this.players[playerIndex];
    for (let i = 0; i < count; i++) {
        if (this.deck.length === 0) {
            this.reshuffleDeck();
        }
        if (this.deck.length > 0) {
            player.hand.push(this.deck.pop());
        }
    }
}

drawCard(playerId) {
    const playerIndex = this.players.findIndex(p => p.id === playerId);
    if (playerIndex === -1 || playerIndex !== this.currentPlayer) {
        return { success: false, error: 'Not your turn' };
    }

    // Prevent drawing more than once per turn
    if (this.hasDrawnThisTurn) {
        return { success: false, error: 'You can only draw once per turn' };
    }

    if (this.stackedDrawCount > 0) {
        const count = this.stackedDrawCount;
        this.drawCards(playerIndex, this.stackedDrawCount);
        this.stackedDrawCount = 0;
        this.stackStartedBy = null;
        this.advanceTurn();
        return { success: true, drewStacked: true, stackCount: count };
    }
```

```
        const playerHand = this.players[playerIndex].hand;

        // Always draw exactly one card per turn
        this.hasDrawnThisTurn = true;
        this.drawCards(playerIndex, 1);

        const drawnCard = playerHand[playerHand.length - 1];
        const canPlay = drawnCard ? this.canPlayCard(drawnCard) : false;

        if (!canPlay) {
            this.advanceTurn();
        }

        return { success: true, canPlayDrawn: canPlay, cardsDrawn: 1 };
    }

    callUno(playerId) {
        const playerIndex = this.players.findIndex(p => p.id === playerId);
        if (playerIndex === -1) {
            return { success: false, error: 'Player not found' };
        }

        const player = this.players[playerIndex];

        // Can only call O,no when player has 2 or 1 cards
        if (player.hand.length <= 2 && player.hand.length > 0) {
            player.calledUno = true;
            return { success: true, playerName: player.name };
        }

        return { success: false, error: 'Can only call O,no with 2 or 1 cards' };
    }

    catchUnoViolation(catcherId, caughtPlayerId) {
        const catcherIndex = this.players.findIndex(p => p.id === catcherId);
        const caughtIndex = this.players.findIndex(p => p.id === caughtPlayerId);

        if (catcherIndex === -1 || caughtIndex === -1) {
            return { success: false, error: 'Player not found' };
        }

        const caughtPlayer = this.players[caughtIndex];

        // Check if the caught player has 1 card and hasn't called O,no
        if (caughtPlayer.hand.length === 1 && !caughtPlayer.calledUno) {
            // Apply penalty: draw 2 cards
```

```
            this.drawCards(caughtIndex, 2);
            caughtPlayer.calledUno = false; // Reset the flag

            return {
                success: true,
                catcherName: this.players[catcherIndex].name,
                caughtName: caughtPlayer.name,
                penaltyApplied: true
            };
        }

        return { success: false, error: 'No O,no violation detected' };
    }

    checkUnoAfterPlay(playerIndex) {
        const player = this.players[playerIndex];

        // If player has exactly 1 card after playing and didn't call O,no, apply
        if (player.hand.length === 1 && !player.calledUno) {
            this.drawCards(playerIndex, 2);
            return { penaltyApplied: true, playerName: player.name };
        }

        // Reset calledUno flag when hand length is no longer 1
        // (player has won with 0 cards or has more than 1 card)
        if (player.hand.length !== 1) {
            player.calledUno = false;
        }

        return { penaltyApplied: false };
    }

    reshuffleDeck() {
        if (this.discardPile.length <= 1) return;

        const topCard = this.discardPile.pop();
        this.deck = [...this.discardPile];
        this.discardPile = [topCard];
        this.shuffleDeck();
    }
}

/* ======================
   SOCKET.IO
====================== */

io.on('connection', socket => {
```

```
console.log('Connected:', socket.id);

socket.on('requestLobbies', () => {
    socket.emit('lobbyList', Object.values(lobbies));
});

socket.on('createLobby', ({ lobbyName, playerName, settings }) => {
    const id = `lobby_${Date.now()}`;
    const minPlayers = 2; // minimum is always 2 (supports 1v1 up to 8 player

    const presenceManager = new PlayerPresenceManager(minPlayers, {
        heartbeatInterval: 5000,
        reconnectTimeout: 60000
    });
    lobbyPresenceManagers.set(id, presenceManager);

    // Add player to presence manager
    presenceManager.addPlayer(socket.id, playerName, PlayerState.LOBBY);

    lobbies[id] = {
        id,
        name: lobbyName,
        settings,
        players: [{ id: socket.id, name: playerName, ready: false }],
        minPlayers
    };
    socket.join(id);
    socket.emit('lobbyCreated', {
        roomId: id,
        lobbyName: lobbyName,
        settings: settings,
        players: lobbies[id].players,
        minPlayers
    });
    broadcastLobbyList();
});

socket.on('joinLobby', ({ lobbyId, playerName }) => {
    const lobby = lobbies[lobbyId];
    if (!lobby) return;

    // Check if player is already in lobby
    if (lobby.players.some(p => p.id === socket.id)) {
        socket.emit('error', 'You are already in this lobby');
        return;
    }
```

```
    // Check if lobby has reached maxPlayers limit
    if (lobby.players.length >= lobby.settings.maxPlayers) {
        socket.emit('error', 'Lobby is full');
        return;
    }

    // Add to presence manager
    const presenceManager = lobbyPresenceManagers.get(lobbyId);
    if (presenceManager) {
        presenceManager.addPlayer(socket.id, playerName, PlayerState.LOBBY);
    }

    lobby.players.push({ id: socket.id, name: playerName, ready: false });
    socket.join(lobbyId);
    socket.emit('lobbyJoined', {
        roomId: lobbyId,
        lobbyName: lobby.name,
        settings: lobby.settings,
        players: lobby.players,
        minPlayers: lobby.minPlayers || 2
    });
    io.to(lobbyId).emit('lobbyUpdate', { roomId: lobbyId, players: lobby.play
    broadcastLobbyList();
});

socket.on('playerReady', ({ roomId, ready }) => {
    const lobby = lobbies[roomId];
    if (!lobby) return;

    const player = lobby.players.find(p => p.id === socket.id);
    if (player) player.ready = ready;

    // Update presence manager
    const presenceManager = lobbyPresenceManagers.get(roomId);
    if (presenceManager) {
        presenceManager.setPlayerReady(socket.id, ready);
    }

    // Send lobby update to all players in the lobby
    io.to(roomId).emit('lobbyUpdate', { roomId: roomId, players: lobby.player

    const minPlayers = lobby.minPlayers || 2;

    // Check if can start game
    if (lobby.players.length >= minPlayers && lobby.players.every(p => p.read
        // Verify all players are still connected
        const allConnected = lobby.players.every(p => !!io.sockets.sockets.ge
```

```
if (!allConnected) {
    // Remove disconnected players
    lobby.players = lobby.players.filter(p => io.sockets.sockets.get(
    io.to(roomId).emit('lobbyUpdate', { roomId: roomId, players: lobb
    io.to(roomId).emit('error', 'Some players disconnected before gam
    return;
}

// Final presence check using presence manager
if (presenceManager) {
    const presenceCheck = presenceManager.finalPresenceCheck();
    if (!presenceCheck.success) {
        const missingNames = presenceCheck.missingPlayers.map(p => p.
        io.to(roomId).emit('error', `Cannot start: Players not respon
        console.log(`[Server] Game start prevented - missing players:
        return;
    }
}

console.log(`[Server] Starting game in lobby ${roomId} with ${lobby.p

const room = new GameRoom(roomId, lobby.players, lobby.settings);
rooms.set(roomId, room);
room.createDeck();
room.dealCards();
room.gameStarted = true;

// Move players to IN_GAME state
if (presenceManager) {
    lobby.players.forEach(p => {
        presenceManager.updatePlayerState(p.id, PlayerState.IN_GAME);
    });
    // Start heartbeat monitoring for in-game presence
    presenceManager.startHeartbeat();

    // Set up timeout handler for the game
    presenceManager.on('player-timeout', (timedOutPlayer) => {
        const room = rooms.get(roomId);
        if (!room) return;

        const activePlayers = room.players.filter(p => {
            const pm = presenceManager.getPlayer(p.id);
            return pm && pm.state !== PlayerState.TIMEOUT && pm.state
        });

        if (activePlayers.length < minPlayers) {
```

```
                    // Not enough players, end game
                    io.to(roomId).emit('gameEnded', {
                        reason: 'Not enough players remaining',
                        message: `${timedOutPlayer.name} disconnected and did
                    });
                    rooms.delete(roomId);
                    presenceManager.destroy();
                    lobbyPresenceManagers.delete(roomId);
                } else {
                    // Continue game with remaining players
                    io.to(roomId).emit('playerTimeout', {
                        playerId: timedOutPlayer.id,
                        playerName: timedOutPlayer.name,
                        message: `${timedOutPlayer.name} has been removed fro
                    });
                }
            });
        }

        room.players.forEach(p => {
            io.to(p.id).emit('gameStarted', room.getGameState(p.id));
        });

        delete lobbies[roomId];
        broadcastLobbyList();
    } else if (lobby.players.length < minPlayers && lobby.players.every(p =>
        // Not enough players
        io.to(roomId).emit('error', `Need at least ${minPlayers} players to s
    }
});

socket.on('playCard', ({ roomId, cardIndex, chosenColor }) => {
    const room = rooms.get(roomId);
    if (!room || !room.gameStarted) return;

    const result = room.playCard(socket.id, cardIndex, chosenColor);

    if (!result.success) {
        socket.emit('error', result.error);
        return;
    }

    // Broadcast draw animation if a +2 or +4 card was played (non-stacking,
    if (result.drawAnimation) {
        io.to(roomId).emit('drawAnimation', result.drawAnimation);
    }
```

```
    // Broadcast updated game state to all players
    room.players.forEach(p => {
        io.to(p.id).emit('gameState', room.getGameState(p.id));
    });

    // Broadcast O,no penalty if applied
    if (result.unoPenalty) {
        io.to(roomId).emit('unoPenalty', {
            playerName: result.unoPenalty.playerName,
            message: `${result.unoPenalty.playerName} forgot to call O,No! Dr
        });
    }

    // Check for winner
    if (result.winner !== null) {
        const winnerName = room.players[result.winner].name;
        const winnerId = room.players[result.winner].id;
        io.to(roomId).emit('gameOver', { winner: winnerName, winnerId });
        rooms.delete(roomId);
    }
});

socket.on('drawCard', ({ roomId }) => {
    const room = rooms.get(roomId);
    if (!room || !room.gameStarted) return;

    const result = room.drawCard(socket.id);

    if (!result.success) {
        socket.emit('error', result.error);
        return;
    }

    // Emit draw animation if player absorbed a stacked draw
    if (result.drewStacked) {
        const playerData = room.players.find(p => p.id === socket.id);
        io.to(roomId).emit('drawAnimation', {
            victimId: socket.id,
            victimName: playerData?.name,
            playerId: null,
            count: result.stackCount || 2,
            cardValue: 'stack'
        });
    }

    // Broadcast updated game state to all players
    room.players.forEach(p => {
```

```
        const state = room.getGameState(p.id);
        // Add draw info to the game state if cards were drawn
        if (result.cardsDrawn) {
            state.lastDrawInfo = {
                cardsDrawn: result.cardsDrawn,
                playerId: socket.id
            };
        }
        io.to(p.id).emit('gameState', state);
    });
});

socket.on('callUno', ({ roomId }) => {
    const room = rooms.get(roomId);
    if (!room || !room.gameStarted) return;

    const result = room.callUno(socket.id);

    if (result.success) {
        // Broadcast to all players that this player called O,no
        io.to(roomId).emit('playerCalledUno', {
            playerName: result.playerName,
            message: `${result.playerName} called O,No!`
        });
    }
});

socket.on('catchUno', ({ roomId, caughtPlayerId }) => {
    const room = rooms.get(roomId);
    if (!room || !room.gameStarted) return;

    const result = room.catchUnoViolation(socket.id, caughtPlayerId);

    if (result.success && result.penaltyApplied) {
        // Broadcast penalty to all players
        io.to(roomId).emit('unoPenalty', {
            catcherName: result.catcherName,
            caughtName: result.caughtName,
            message: `${result.catcherName} caught ${result.caughtName}! ${re
        });

        // Broadcast updated game state to all players
        room.players.forEach(p => {
            io.to(p.id).emit('gameState', room.getGameState(p.id));
        });
    } else if (!result.success) {
        socket.emit('error', result.error);
```

```
    }
});

// Heartbeat handler - clients send this every 5 seconds
socket.on('heartbeat', ({ roomId }) => {
    // Check if player is in a lobby
    const lobby = lobbies[roomId];
    if (lobby) {
        const presenceManager = lobbyPresenceManagers.get(roomId);
        if (presenceManager) {
            presenceManager.receiveHeartbeat(socket.id);
        }
    }

    // Check if player is in an active game
    const room = rooms.get(roomId);
    if (room) {
        const presenceManager = lobbyPresenceManagers.get(roomId);
        if (presenceManager) {
            presenceManager.receiveHeartbeat(socket.id);
        }
    }
});

socket.on('disconnect', () => {
    console.log('Disconnected:', socket.id);

    // Handle lobby disconnections
    Object.keys(lobbies).forEach(id => {
        const hadPlayer = lobbies[id].players.some(p => p.id === socket.id);

        if (hadPlayer) {
            const presenceManager = lobbyPresenceManagers.get(id);
            if (presenceManager) {
                presenceManager.onPlayerDisconnect(socket.id);
                const player = presenceManager.getPlayer(socket.id);

                // Notify other players
                if (player) {
                    io.to(id).emit('playerDisconnected', {
                        playerId: socket.id,
                        playerName: player.name,
                        reconnectTimeout: presenceManager.reconnectTimeout
                    });
                }
            }
```

```
                lobbies[id].players = lobbies[id].players.filter(p => p.id !== so

                // Notify remaining players in the lobby if someone left
                if (lobbies[id].players.length > 0) {
                    io.to(id).emit('lobbyUpdate', { roomId: id, players: lobbies[
                } else {
                    // Clean up empty lobby
                    delete lobbies[id];
                    if (presenceManager) {
                        presenceManager.destroy();
                        lobbyPresenceManagers.delete(id);
                    }
                }
            }
        });

        // Handle in-game disconnections
        rooms.forEach((room, roomId) => {
            const playerIndex = room.players.findIndex(p => p.id === socket.id);
            if (playerIndex !== -1) {
                const player = room.players[playerIndex];
                const presenceManager = lobbyPresenceManagers.get(roomId);

                if (presenceManager) {
                    presenceManager.onPlayerDisconnect(socket.id);

                    // Notify other players in game
                    room.players.forEach(p => {
                        if (p.id !== socket.id) {
                            io.to(p.id).emit('playerDisconnected', {
                                playerId: socket.id,
                                playerName: player.name,
                                reconnectTimeout: presenceManager.reconnectTimeou
                            });
                        }
                    });
                }
            }
        });

        broadcastLobbyList();
    });
});

/* ======================
   HELPERS
====================== */
```

```
function broadcastLobbyList() {
    io.emit('lobbyList', Object.values(lobbies));
}

/* ======================
   START SERVER
====================== */

server.listen(PORT, () => {
    console.log(`O,No server running on port ${PORT}`);
});
```