How-To Guide: Colyseus

[Memotrace]

Owen Post

April 11, 2025

1 Introduction

This guide explains how to use **Colyseus** to build a multiplayer game server, highlighting key implementation aspects from our project. We cover setting up a game room with state transitions, UDP communication for external devices, and client-side integration. While other guides exist for setting up and using Colyseus, this guide details how to use it with UDP and lays out everything in easy-to-follow steps.

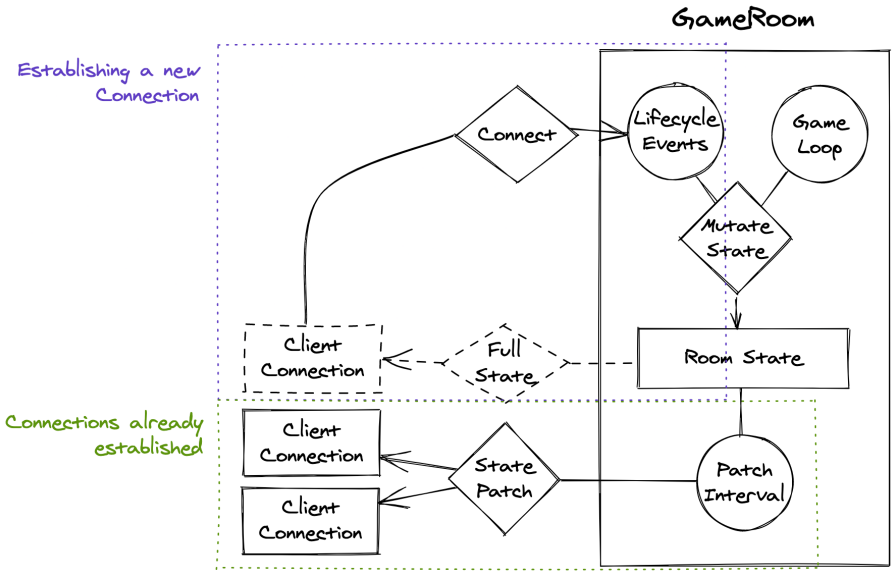

Figure 1: Colyseus GameRoom Cycle

Colyseus is a framework built on Node.js and WebSockets that simplifies real-time networking for multiplayer applications. Each client connects to a specific room on the server using a WebSocket based on the room's unique identifier. When a new user joins a room, they receive a complete copy of the current server state. After this initial synchronization, the server sends incremental updates that allow each client to continuously update their local state and maintain synchronized rendering of the application.

In our application, we use Colyseus to streamline both individual calls—such as tracking user movements—and global calls, like starting the game for all users in a room.

2 Required Tools and Setup

– **Node.js** and **npm** for running the server.

– **Colyseus** framework.

– **dgram** module for UDP communication.

– A modern web browser for client testing.

3 Step-by-Step Guide

Step 0.5: Pre-setup Steps (Skip if you already have your server files)

First, download and install Node.js and npm if you haven't already. These tools are required to run a web server and manage dependencies. You can skip this step if Node.js and npm are already installed on your system.

```
sudo apt update
sudo apt install nodejs npm
```

If you don't already have a server directory for your project, create one:

```
mkdir web-server
cd web-server
```

Next, initialize the project with npm. This will create a `package.json` file to manage project dependencies.

```
npm init -y
```

Now, set up the basic folder structure:

```
mkdir public
mkdir lib
```

- The `public` folder will contain all frontend assets such as HTML, CSS, and JavaScript files.
- The `lib` folder will contain your server-side game logic, including Colyseus room classes and handlers.

Your main server entry point will be a file named `server.js`, located at the root of the `web-server` directory.

To start the server, simply run:

```
node server.js
```

Step 1: Setting Up the Server Environment

Now, install the necessary packages:

```
npm install colyseus express http dgram
```

Listing 1: Necessary Packages

These are the tools we will need for setting up your Colyseus server and UDP communication.

Step 2: Creating the `server.js` File

In this step, we'll set up the `server.js` file, which serves as the backbone of our application. Once configured, you will rarely need to modify this file.

1. Import Required Packages

At the top of `server.js`, include the packages required for our server setup. These modules handle HTTP requests, serve static files, manage file paths, and integrate Colyseus for our multiplayer functionality.

```
const http = require("http");
const express = require("express");
const path = require("path");
const colyseus = require("colyseus");
const { GameRoom } = require("./lib/game_server");
```

Listing 2: Importing Required Packages

2. Configure Express and the HTTP Server

Next, create an instance of an Express application and define a port number (in this example, 8088). Then, instruct Express to serve static files (like HTML, CSS, and JavaScript) from the `public` directory. Finally, create an HTTP server that uses the Express app to handle requests.

```
const app = express();
const PORT_NUMBER = 8088;

// Serve static files from the "public" folder.
app.use(express.static(path.join(__dirname, "public")));

// Create an HTTP server using the Express app.
const server = http.createServer(app);
```

Listing 3: Configuring Server

3. Integrate Colyseus

Now, create a Colyseus game server instance by passing the HTTP server to it. This integration allows Colyseus to handle real-time multiplayer functionalities alongside your web server. Then, define a new room named "game" that uses our custom `GameRoom` class (which will be defined later in `lib/game_server.js`).

```
const gameServer = new colyseus.Server({ server });

gameServer.define("game", GameRoom);
```

Listing 4: Adding Colyseus

4. Start the Server

Finally, start the HTTP server on the designated port. When the server is running, a message will be logged to the console indicating that it's listening on the specified port.

```
server.listen(PORT_NUMBER, () => {
  console.log("Server listening on port", PORT_NUMBER);
});
```

Listing 5: Server Start

Step 3: Implementing the Game Room

In this step, we create a new file in the `lib/` directory named `game_server.js`. In this file, we define a `GameRoom` class that extends Colyseus's `Room` class. This class will manage room events (joining and leaving) as well as handle UDP communication with our backend.

Basic Structure Start by requiring the necessary modules and defining an empty `GameRoom` class:

```
const colyseus = require("colyseus");
const dgram = require("dgram");

class GameRoom extends colyseus.Room {
  // Room lifecycle methods and UDP communication will be added here.
}

module.exports = { GameRoom };
```

Listing 6: Basic structure of game_server.js

Room Lifecycle Methods Colyseus rooms have built-in lifecycle methods such as `onCreate`, `onJoin`, and `onLeave`. We can use these to initialize our UDP socket and log room events. For example:

```
onCreate(options) {
  // Create a UDP socket to communicate with the backend.
  this.udpClient = dgram.createSocket("udp4");
  console.log("GameRoom created.");
}

onJoin(client, options) {
  // Log the client's sessionId upon joining.
  console.log("Client joined:", client.sessionId);
}

onLeave(client, consented) {
  // Log the client's sessionId upon leaving.
  console.log("Client left:", client.sessionId);
}
```

Listing 7: Room lifecycle methods

Defining a UDP Relay Function Next, we define a helper function called `sendThroughUDP` within the `GameRoom` class. This function simplifies sending data via UDP and handling the response. In this example, we specify a target PORT and IP address (adjust these as needed), convert the data into a Buffer, and send it. We also set up a listener for a single response message:

```
sendThroughUDP(data, callback) {
  const PORT = 12345;
  const HOST = "192.168.7.2";
  const buffer = Buffer.from(data);

  // Send the data over UDP.
  this.udpClient.send(buffer, 0, buffer.length, PORT, HOST, (err) => {
    if (err) {
      console.error("UDP send error:", err);
      callback("SERVER ERROR: Send failed");
    }
```

```
  });

  // Listen for a single incoming message.
  // (Note: If no response is expected, this section may need to be modified.)
  this.udpClient.once("message", (message, remote) => {
    callback(message.toString());
  });
}
```

Listing 8: sendThroughUDP function

Handling Incoming Messages Finally, within `onCreate` we set up an `onMessage` handler for room-specific commands. For example, when the room receives a "start" message from a client, we use the `sendThroughUDP` function to relay a "start" command to our backend via UDP. The response from the UDP call can then be processed as needed. Another helpful method is `broadcast`, this can be useful for notifying all connected clients about a global event without sending individual messages.

```
onCreate(options) {
  this.udpClient = dgram.createSocket("udp4");
  console.log("GameRoom created.");

  // Set up a message handler for room messages.
  this.onMessage("start", (client) => {
    console.log("Start message received from:", client.sessionId);
    this.sendThroughUDP("start", (response) => {
      // Process the UDP response.
      console.log("Received UDP response:", response);
      // Optionally, broadcast the response to all clients.
      this.broadcast("start-reply", response);
    });
  });
}
```

Listing 9: Handling a "start" command via UDP

Step 4: Integrating Colyseus into Your Frontend JavaScript

With our server-side files in place, we can now integrate Colyseus into the frontend JavaScript. If you don't already have one, create a `game_logic.js` file in your public folder, along with a corresponding `game.html` file. You're free to design any HTML frontend you'd like, but for now, we'll focus on adding the necessary JavaScript code to `game_logic.js` to establish a connection with the Colyseus server.

At the top of the file, initialize the connection by creating a new Colyseus client. Make sure to replace the port number (8088) with the one defined in your `server.js`.

```
const client = new Colyseus.Client(`ws://${window.location.hostname}:8088`);
```

Listing 10: Connecting to the Colyseus Server

Next, use the client to join or create a game room. Each room has a unique identifier (`room.id`), and you can send messages to the backend by calling `room.send(message)`. Likewise, you can register handlers for different message types that the server sends back. For example, to initiate the game with a "start" command and listen for a "start-reply", you can use:

```
client
  .joinOrCreate("game")
  .then((room) => {
```

```
    console.log("Joined room:", room.id);

    // Send a "start" message to trigger UDP communication in our GameRoom.
    room.send("start");

    // Listen for the server's response to the "start" message.
    room.onMessage("start-reply", (message) => {
      console.log("Received start reply:", message);
    });
  })
  .catch(console.error);
```

Listing 11: Joining a Room and Sending/Receiving Messages

If you want to reference the room outside of the join/create callback (for instance, to send additional messages later), declare a global variable and assign the room to it:

```
let gRoom;

client
  .joinOrCreate("game")
  .then((room) => {
    gRoom = room;
    console.log("Joined room:", room.id);
  })
  .catch(console.error);

// Later in your code, you can use:
gRoom.send("start");
```

Listing 12: Storing the Room Reference

4 Troubleshooting

- **UDP Timeouts:** If you see "UDP timeout" errors, verify the external device is connected at the correct IP and port.
- **UDP Timeouts or Early Disconnection:** Check if you properly handle all replies. If you set up your code to expect replies from your UDP socket, then every call must have a reply sending some kind of data back.
- **Getting the Same Data returned from all UDP calls:** If you use an interval to repeatedly send UDP requests to your backend and receive responses, make sure to schedule each callback on a separate interval. Batching UDP calls together on the same interval can cause duplicate and mixed-up data responses.

References

- Colyseus Documentation: `https://docs.colyseus.io/`
- Node.js Documentation: `https://nodejs.org/en/docs/`
- Figure 1 from: `https://www.imini.app/docs/tutorial-multiple-player/server-colyseus/`