# *Web Programming III (420-H30-HR)*

# *Assignment 2 – Multiplayer Pong Game*

Date assigned: October 31, 2023

Date due: **Friday, November 17, 2023**

**Learning Objectives**

Upon successful completion of this assignment, the student will have:

* Converted a single player pong game to a multiplayer game
* Understood the concept of rooms in socket.io

General idea of the assignment:

The assignment file contains the code for a single player pong game. The task is to refactor it into a multiplayer game by leveraging the socket.io library. In this assignment, you will be given step by step instructions of how to accomplish this. In addition, a sequence diagram showing the sequence of interactions between the client and server is attached to the assignment folder on Moodle.

To do:

1. **Setting up a socket.io server**:
   1. Download and extract the starting folder (**single-player-pong**). Rename this to <username>\_multiplayer\_pong
   2. Open the folder in VSCode (or your preferred IDE). In the terminal, type **npm init -y** in the root folder. Your package.json should look similar to the following:

A screenshot of a computer code

Description automatically generated

* 1. Using npm, also install socket.io, express and nodemon
  2. Create the **server.js** file at the same level as the package.json file
     1. Create a simple http server that listens on port 3000.

const http = require(‘http’)

const httpServer = http.createServer();

* + 1. Require socket.io and attach it to the server you just created

const io = require(‘socket.io’)(httpServer, {

cors: {

origin: “\*”,

methods: [‘GET’, ‘POST’]

}

});

* + 1. Add an event listener to the **io** object (created above) using the on() method that listens for ‘connection’ – the event fired when we get a new connection. In the callback function, pass in a parameter called **socket** to communicate with the client. For now, you can simply console.log the message `A user connected` inside the callback function.
  1. Test that your server is working by running the server.js file using nodemon. At this point, it should only display output in the console terminal (not browser)

1. **Connecting to socket.io**: The goal of this step is to connect the index.html (client) to the server we just created thereby making it full stack.
   1. In the index.html file, paste the following script tag above the line that has a <script> tag:

**<script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.7.2/socket.io.js" integrity="sha512-zoJXRvW2gC8Z0Xo3lBbao5+AS3g6YWr5ztKqaicua11xHo+AvE1b0lT9ODgrHTmNUxeCw0Ry4BGRYZfXu70weg==" crossorigin="anonymous" referrerpolicy="no-referrer"></script>**

* 1. In the script.js file, connect to the server by adding the following line of code below the line that begins with ‘const context’ (line 3)

const socket = io(‘http://localhost:3000’)

* 1. Since we want to make the game multiplayer, we can now clean up the **script.js** file by removing all references to the computer player (i.e. lines 27, 150-153, 160-174, 178).
  2. Still in the **script.js** file, uncomment the **renderIntro()** function and the line where it is invoked
  3. Test the connection by opening the index.html file using LiveServer. You should see a message displayed on the VS Code terminal saying, “**a user connected**”.

1. **Identifying connected clients**:
   1. Next, we implement the first step of the sequence diagram which is emit(‘ready’)
      1. In script.js, inside the startGame() function, add the following line below the renderIntro() invocation line:

socket.emit(‘ready’)

* 1. The line above represents the information sent by the client (player) to the server notifying the server that the player is ready to play.
  2. In script.js, below the startGame() method invocation, we add a ‘connect’ event listener to our socket like the following:

socket.on(‘connect’, () => {

console.log(`Connected as … ${socket.id}`)

})

* 1. Similarly, on the server side (server.js), we modify the io.on(‘connection’..) to include the id of the socket (i.e. connected client) as follows:

io.on(‘connection’, (socket) => {

console.log(`A user connected with id ${socket.id}`)

})

* 1. Test that user id logged on the IDE terminal matches that of the browser console

1. **Listening for events in the Pong Server**: At this point, the player(s) (client) have notified the server of their readiness. The server now needs to listen and confirm that two players have confirmed their readiness and also respond to the ‘ready’ event by executing the ‘startGame’ event
   1. Declare a variable called **readyPlayerCount** and initialize it to zero just before the io.on(‘connection’, (socket)=> {…}) code in server.js
   2. Inside the ‘connection’ event on the server add an event listener to the socket object to handle ‘ready’ event as follows:

socket.on(‘ready’, () => {

console.log(`Player with id ${socket.id} ready`)

})

* 1. Still inside the ready event handler add a line right after the console.log code to increment the readyPlayerCount by 1.
  2. Next to that line, check if the readyPlayerCount is exactly 2 and then broadcast ‘startGame’ event using io.emit(‘startGame’, socket.id). This will broadcast to all connected clients (players) and sets the last player to respond as the referee. The referee role will keep track of the ball position, score and emit the same to the connected clients.

1. **Handling events in the Pong client**: At this point, the server has sent each client a startGame event. We want to handle this on the client side (script.js)
   1. Add a handler below the ‘connect’ event handler to handle the ‘startGame’ event sent from the server. The event should also take in the **refereeId** (i.e. the second player to show readiness)

socket.on(‘startGame’, (refereeId) => {

console.log(`Referee is ${refereeId}`)

})

* 1. At the top of the **script.js** file create a Boolean variable called **isReferee** that is initially set to **false**. It is meant to determine whether the current client is the referee.
  2. Back in the ‘startGame’ event handler in script.js, add a line of code to set the value of **isReferee** based on whether the current client’s id (socket.id) matches the id returned by the server (refereeId).
  3. Next, we will split the startGame() function in script.js into two functions: loadGame() and startGame(). As a result, loadGame() and startGame() will now look like the following screenshot:

A screen shot of a computer code

Description automatically generated

* 1. Below the line of code written in step 5c, invoke the **startGame()** function.
  2. In the first line of the startGame() function, observe the paddleIndex which is set to 0 which represents the position of the player 1 when the game was single player. Now that we are making it multiplayer, we want the referee to be at the bottom position (i.e. 0) while the second player is at the top position (i.e. 1). Refactor the line of code thus:

paddleIndex = isReferee ? 0 : 1;

* 1. Test that the game at this point to ensure that it is loading correctly and that the referee is also getting assigned. This can be done by opening two browsers side by side. Try switching between the two browsers by moving your mouse between each and clicking in. Do you observe how the two screens seem out of sync? We will fix this in step 6.
  2. Lastly, open the two browser consoles and see the output. Compare it with the console output in the server (VS Code terminal)

1. Implementing the game logic for Paddle: In this step, we will synchronize the two browser screens especially the paddle positions. The position of the two paddles is tracked in the array paddleX = [255, 255]
   1. In the startGame() function of script.js, below the two if-statements, add the following line of code:

socket.emit(‘paddleMove’, {

xPosition: paddleX[paddleIndex]

})

This will emit the position of the current player (paddle) to the server so that it can then broadcast it to both clients

* 1. In server.js, inside the **io.on(‘connection’, ()=> {…})** event listener, add another event that listens and responds to ‘paddleMove’.

socket.on(‘paddleMove’, (paddleData) => {

socket.broadcast.emit(‘paddleMove’, paddleData)

})

This will receive paddleData from the clients and broadcast to all connected players except the sender.

* 1. On the client side (script.js), we want to be able to listen for the ‘paddleMove’ event being broadcast by the server to synchronize the paddle positions. We achieve this by adding the following code:

socket.on(‘paddleMove’, (paddleData) => {

// Toggle 1 to 0, and 0 to 1

const opponentPaddleIndex = 1 – paddleIndex;

paddleX[opponentPaddleIndex] = paddleData.xPosition;

})

* 1. Test the program again by launching two browsers. You will observe that even though the two browsers are ready, the game does not yet start. For now, simply refresh the browsers manually to get it going. Once this is done, observe that the paddles are now in sync on both browsers.

1. **Implementing the game logic for Ball**: Next, we need to ensure the balls are also synchronized. We achieve this by synchronizing the ballData.
   1. In the ballMove() function of script.js, add a line of code after the if statement to emit the ‘ballMove’ event and pass in an object containing **ballX** and **ballY**. In addition, we add the **score** array as follows:

socket.emit(‘ballMove’, {

ballX,

ballY,

score

})

* 1. Add the same code in the ballReset() function of script.js as well.
  2. The referee player will be the source of truth for calculating and sharing the ballData info. Therefore, in the animate() function of script.js, modify the code to only execute ballMove() and ballBoundaries() if the player is the referee (i.e. isReferee is true). The renderCanvas() function should execute afterwards.
  3. In server.js, inside the io.on(‘connection’, (socket) => {}) code block, add a line to handle the ballMove event emitted by the client

socket.on(‘ballMove’, (ballData) => {

socket.broadcast.emit(‘ballMove’, ballData)

})

* 1. Back in script.js, add code below the line **socket.on(‘paddleMove’, (paddleData) => {…})** to handle broadcast emit ‘ballMove’ event coming from the server like so:

socket.on(‘ballMove’, (ballData) => {

({ballX, ballY, score} = ballData);

})

* 1. Test the program now by opening two browsers side by side. You can launch the index.html file using LiveServer. The ball movement and score in both browsers should now be in sync.

1. **Handling disconnections**: Next, what would happen if one of the players decided to disconnect?
   1. In server.js, add a listener for **disconnect** event inside the io.on(‘connection’, (socket) => {}) block like so:

socket.on(‘disconnect’, (reason) => {

console.log(`Client ${socket.id} disconnected: ${reason}`);

})

If you try to run the game after a player has disconnected you observe that the game never resumes even after a player has rejoined the game. This can be fixed through the concept of “rooms” in socket.io. We will come to that shortly. In the meantime, to fix the issue where the game refuses to resume after a player disconnects, do the following:

* 1. In server.js, locate the condition where we hardcoded **readyPlayerCount === 2** and replace it even player count as follows:

readyPlayerCount % 2 === 0

* 1. The game should now be able to resume playing

1. **Using socket.io with Express**: Since socket.io integrates well with Express, we will now refactor our code to introduce Express into the project.
   1. Create a folder called **public** at the root directory of your project.
   2. Move the **index.html** file into the public folder. In addition, also move the **javascripts** and **stylesheets** folders into the public folder.
   3. Create a file called **api.js** at the same level as your server.js file.
   4. Inside the api.js file,
      1. Require express module and assign it to a ‘const’ant called **express**
      2. Require the path module and assign it to a constant called path
      3. Create another constant called api which is an instantiation of the express module (i.e. const api = express())
      4. The express object (i.e. api) should use the express middleware (express.static()) to read from the folder called public
      5. Express should serve the index.html file whenever there is a get request at the root route (/).
      6. Express should listen on port 3001
      7. Test that this set up works by running the api.js file. You should be able to go to the browser type <http://localhost:3001> and see the game being served
   5. To ensure that we can run our server.js and api.js using only port 3000. We will make the following adjustments to our code:

Create a new file called **sockets.js** at the same level as server.js in your project folder

Inside socket.js create an empty function block called **listen** that takes **io** as parameter

function listen(io) {}

Export the **listen** function using module.exports

In api.js file

Export the **api** object using module.exports

Comment out the api.listen(3001) line

In server.js file:

Copy out the io.on(‘connection’, (socket) => {…}) code block into the empty listen function created in the sockets.js file

Also copy out the readyPlayerCount variable in server.js and paste it above the listen function in sockets.js

Require the sockets file you created. You can assign it to a constant called sockets

Call the listen method on sockets and pass in the io parameter

sockets.listen(io)

On the very first line of server.js, require the api.js file and assign it to a constant called **apiServer**. This represents the express object that was exported in the api.js file

Pass **apiServer** into the http createServer line as follows:

const httpServer = http.createServer(apiServer)

Refactor the following code from:

const io = require(‘socket.io’)(httpServer, {

cors: {

origin: “\*”,

methods: [‘GET’, ‘POST’]

}

});

To this:

const io = require(‘socket.io’)

const socketServer = io(httpServer);

Modify the line sockets.listen(io) to sockets.listen(socketServer)

After making all the above changes, you should now be able to run the application from <http://localhost:3000> by simply running the server.js file

1. **Namespace**: A namespace is a communication channel that allows you to split the logic of your application over a single shared connection. To add it to our application we will perform the following steps:
   1. In script.js
      1. Modify the line **const socket = io(‘http://localhost:3000’)** to the following:

const socket = io(‘/pong’)

* 1. In sockets.js,
     1. Add the following line as the first line of code in the listen function

function listen(io) {

const pongNamespace = io.of(‘/pong’);

// rest of the listen code previously written

}

* + 1. Replace all references to **io** (total of 2) with pongNamespace inside the listen function

1. **Introducing Rooms to the game**: Rooms are an advanced feature that come with socket.io. They provide support for multiple players joining a game. Rooms allow us to split socket.io namespaces into groupings of sockets.
   1. In sockets.js do the following:
      1. Create a variable called room inside the pongNamespace.on(‘connection’, (socket) => {}) code block like so:

pongNamespace.on(‘connection’, (socket) => {

let room;

// previously written code

})

* + 1. Inside the socket.on(‘ready’) event listener, write code to join a room like so:

socket.on(‘ready’, () => {

room = ‘room’ + Math.floor(readyPlayerCount / 2)

socket.join(room);

console.log(`Player with id ${socket.id} ready ${room}`)

// Codes previously written

})

* + 1. To leave a room, we add the following code to the socket.on(‘disconnect’) event listener like so:

socket.on(‘disconnect’, (reason) => {

// code previously written

socket.leave(room);

})

* + 1. Instead of broadcasting to all clients on the server when the ‘ready’ event listener is executed, we modify the if statement block from:

if(readyPlayerCount % 2 === 0) {

pongNamespace.emit(‘startGame’, socket.id)

}

To

if(readyPlayerCount % 2 === 0) {

pongNameSpace.in(room).emit(‘startGame’, socket.id)

}

* + 1. We will do a similar thing for our paddleMove and ballMove events as well by replacing broadcast with to(room):

socket.on(‘paddleMove’, (paddleData) => {

socket.to(room).emit(‘paddleMove’, paddleData)

})

And

socket.on(‘ballMove, (ballData) => {

socket.to(room).emit(‘ballMove, ballData)

})

* + 1. After making the above modifications, you should now have a fully multiplayer pong game that allows two pair of players to join a room and start a game.
  1. After making the modifications in sockets.js, you should now have a fully multiplayer pong game that allows multiple pair of players to join a room and start a game.

1. **Bells and whistles**: To customize the application and make it yours, you can do at least two of the following:
   1. Place an appropriate header on the index.html page
   2. Use TypeScript
   3. Style the game using nice colors
   4. Any glitches you might observe during game play

**To submit**

When you have completed the assignment REMOVE THE node\_modules FOLDER, zip the folder containing all the files for the assignment and copy it to the course page.