Chapter 11

Multiplayer with WebSockets

No matter how challenging we make a single-player game, it will always lack the challenge of competing against another human being. Multiplayer games allow players to either compete against each other or work cooperatively toward a common goal.

Now that we have a working single-player campaign, we will look at how we can add multiplayer support to our RTS game by using the HTML5 WebSocket API.

Before we start adding multiplayer to our game, let's first take a look at some networking basics using the WebSocket API with Node.js.

Using the WebSocket API with Node.js

The heart of our multiplayer game is the new HTML5 WebSocket API. Before WebSockets came along, the only way browsers could interact with a server was by polling and long-polling the server with a steady stream of requests. These methods, while they worked, had a very high network latency as well as high-bandwidth usage, making them unsuitable for real-time multiplayer games.

All of this changed with the arrival of the WebSocket API. The API defines a bidirectional, full-duplex communications channel over a single TCP socket, providing us with an efficient, low-latency connection between browser and server.

In simple terms, we can now create a single, persistent connection between a browser and server and send data back and forth much faster than before. You can read more about the benefits of WebSockets at `www.websocket.org/`. Let's take a look at a simple example of communication between the browser and the server using WebSockets.

WebSockets on the Browser

Using WebSockets to communicate with a server involves the following steps:

1. Instantiating a `WebSocket` object by providing the server URL
2. Implementing the `onopen`, `onclose`, and `onerror` event handlers as needed
3. Implementing the `onmessage` event handler to handle actions when a message is received from the server
4. Sending messages to the server by using the `send()` method
5. Closing the connection to the server by using the `close()` method

We can create a simple WebSocket client in a new HTML file, as shown in Listing 11-1. We will place this new file inside the `websocketdemo` folder to keep it separate from our game code.

Listing 11-1. A Simple WebSocket Client (websocketclient.html)

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-type" content="text/html; charset=utf-8">
        <title>WebSocket Client</title>
        <script type="text/javascript" charset="utf-8">

            var websocket;
            var serverUrl = "ws://localhost:8080/";

            function displayMessage(message){
                document.getElementById("displaydiv").innerHTML += message +"<br>";
            }

            // Initialize the WebSocket object and setup Event Handlers
            function initWebSocket(){
                // Check if browser has an implementation of WebSocket (older Mozilla browsers
used MozWebSocket)
                var WebSocketObject = window.WebSocket || window.MozWebSocket;
                if(WebSocketObject){
                    // Create the WebSocket object
                    websocket = new WebSocketObject(serverUrl);

                    // Setup the event handlers
                    websocket.onopen = function(){
                        displayMessage("WebSocket Connection Opened");
                        document.getElementById("sendmessage").disabled = false;
                    };

                    websocket.onclose = function(){
                        displayMessage("WebSocket Connection Closed");
                        document.getElementById("sendmessage").disabled = true;
                    };

                    websocket.onerror = function(){
                        displayMessage("Connection Error Occured");
                    };

                    websocket.onmessage = function(message){
                        displayMessage("Received Message: <i>"+message.data+"</i>");
                    };

                } else {
                    displayMessage("Your Browser does not support WebSockets");
                }
            }
            // Send a message to the server using the WebSocket
            function sendMessage(){
                // readyState can be CONNECTING,OPEN,CLOSING,CLOSED
                if (websocket.readyState = websocket.OPEN){
                    var message = document.getElementById("message").value;
                    displayMessage("Sending Message: <i>"+message+"</i>");
                    websocket.send(message);
                } else {
                    displayMessage("Cannot send message. The WebSocket connection isn't open");
                }
            }


        </script>
    </head>
    <body onload="initWebSocket();">
        <label for="message">Message</label><input type="text" value="Simple Message" size="40" 
id="message">
        <input type="button" value="Send" id="sendmessage" onclick="sendMessage()" disabled="true">
        <div id="displaydiv" style="border:1px solid black;width:600px; height:400px; 
font-size:14px;"></div>
    </body>
</html>
```

The body tag of the HTML file contains a few basic elements: an input box for a message, a button for sending messages, and a div to display all messages.

Within the script tag, we start by declaring a server URL that points to the WebSocket server using the WebSocket protocol (ws://).

We then declare a simple displayMessage() method that appends a given message to the displaydiv div element.

Next we declare the initWebSocket() method that initializes the WebSocket connection and sets up the event handlers.

Within this method, we first check for the existence of the WebSocket or MozWebSocket object to verify that the browser supports WebSockets and save it to WebSocketObject. This is because older versions of the Mozilla browser named their implementation MozWebSocket before shifting to WebSocket.

We then initialize the WebSocket object by calling the constructor of WebSocketObject and saving it to the websocket variable.

Finally, we define handlers for the onopen, onclose, onerror, and onmessage event handlers, where we display appropriate messages to the user. We also enable the sendmessage button when the connection is opened and disable it when the connection is closed.

In the sendMessage() method, we check that connection is open using the readyState property and then use the send() method to send the contents of the message input box to the server.

Our browser client needs a server that it can commmunicate with using the WebSocket protocol. There are already several WebSocket server implementations available for most popular languages such as jWebSocket (http://jwebsocket.org/) and Jetty (http://jetty.codehaus.org/jetty/) for Java, Socket.io (http://github.com/LearnBoost/Socket.IO-node) and WebSocket-Node (https://github.com/Worlize/WebSocket-Node) for Node.js, and WebSocket++ (https://github.com/zaphoyd/websocketpp) for C++.

In this book, we will be using WebSocket-Node for Node.js. We will start by setting up Node.js and creating an HTTP server and then add WebSocket support to it.

Creating an HTTP Server in Node.js

Node.js (http://nodejs.org/) is a server-side platform consisting of several libraries built on top of Google's JavaScript V8 engine. Originally created by Ryan Dahl starting in 2009, Node.js was designed for easily building fast, scalable network applications. Programs are written in JavaScript using an event-driven, nonblocking I/O model that is lightweight and efficient. Node.js has gained a lot of popularity in a relatively short time and is used by a large number of companies, including LinkedIn, Microsoft, and Yahoo.

Before you can start writing Node.js code, you will need to install Node.js on your computer. Implementations of Node.js are available for most operating systems, such as Windows, Mac OS X, Linux, and SunOS, and detailed instructions for setting up Node.js on your specific operating system are available at https://github.com/joyent/node/wiki/Installation. For Windows and Mac OS X, the simplest installation method is to run the ready-made installer files downloadable at http://nodejs.org/download/.

Once Node.js has been set up correctly, you will be able to run Node.js programs from the command line by calling the node executable and passing the program name as a parameter.

After setting up Node.js, we can create a simple HTTP web server inside a new JavaScript file, as shown in Listing 11-2. We will place this file inside the websocketdemo folder.

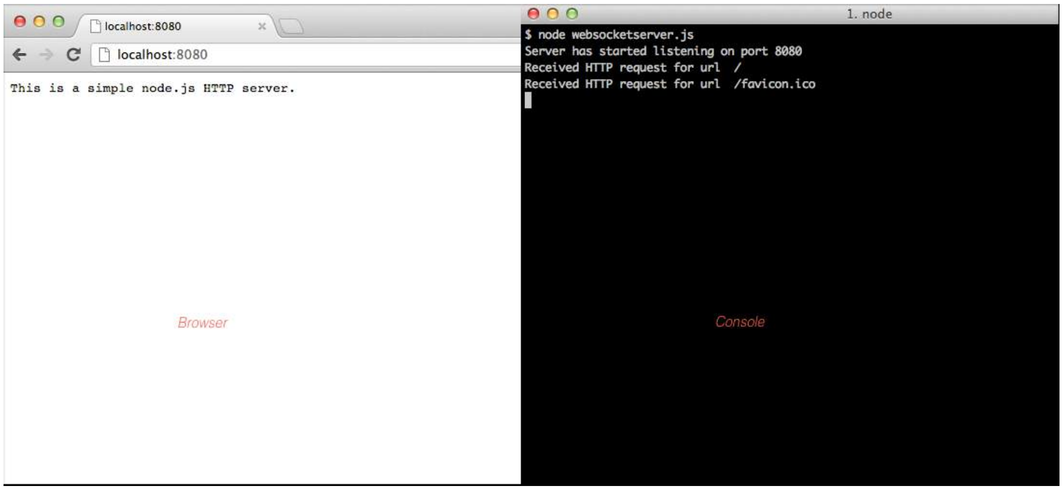
Listing 11-2. A Simple HTTP Web Server in Node.js (websocketserver.js)

```
// Create an HTTP Server
var http = require('http');

// Create a simple web server that returns the same response for any request
var server = http.createServer(function(request,response){
    console.log('Received HTTP request for url ', request.url);
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.end("This is a simple node.js HTTP server.");
});

// Listen on port 8080
server.listen(8080,function(){
    console.log('Server has started listening on port 8080');
});
```

The code for building a simple web server in Node.js is surprisingly small using the Node.js HTTP library. You can find detailed documentation on this library at http://nodejs.org/api/http.html.

We first refer to the HTTP library using the `require()` method and save it to the `http` variable. We then create an HTTP server by calling the `createServer()` method and passing it a method that will handle all HTTP requests. In our case, we send back the same text response for any HTTP request to the server. Finally, we tell the server to start listening on port 8080.

If you run the code in `websocketserver.js` from the command line and try to access the web server's URL (http://localhost:8080) from the browser, you should see the output shown in Figure 11-1.



This is a simple node.js HTTP server.

```
$ node websocketserver.js
Server has started listening on port 8080
Received HTTP request for url  /
Received HTTP request for url  /favicon.ico
```

Figure 11-1. *A simple HTTP server in Node.js*

We have our HTTP server up and running. This server will return the same page no matter what path we pass after the server name in the URL. This server also does not yet support WebSockets.

Next, we will add WebSocket support to this server by using the WebSocket-Node package.

Creating a WebSocket Server

The first thing you will need to do is install the WebSocket-Node package using the `npm` command. Detailed instructions for installation along with sample code is available at `https://github.com/Worlize/WebSocket-Node`.

If Node.js is set up correctly, you should be able to set up WebSocket by running the following command from the command line:

```
npm install websocket
```

■ **Tip** In case you have previously installed Node.JS and WebSocket-Node, you should ensure that you are using the latest version by running the `npm update` command.

Once the WebSocket package has been installed, we will add WebSocket support by modifying `websocketserver.js`, as shown in Listing 11-3.

Listing 11-3. Implementing a Simple WebSocket Server (websocketserver.js)

```
// Create an HTTP Server
var http = require('http');

// Create a simple web server that returns the same response for any request
var server = http.createServer(function(request,response){
  console.log('Received HTTP request for url', request.url);
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end("This is a simple node.js HTTP server.");
});

// Listen on port 8080
server.listen(8080,function(){
  console.log('Server has started listening on port 8080');
});

// Attach WebSocket Server to HTTP Server
var WebSocketServer = require('websocket').server;
var wsServer = new WebSocketServer({
  httpServer:server
});

// Logic to determine whether a specified connection is allowed.
function connectionIsAllowed(request){
  // Check criteria such as request.origin, request.remoteAddress
  return true;
}

// Handle WebSocket Connection Requests
wsServer.on('request',function(request){
  // Reject requests based on certain criteria
  if(!connectionIsAllowed(request)){
     request.reject();
     console.log('WebSocket Connection from' + request.remoteAddress + 'rejected.');
     return;
  }
  // Accept Connection
  var websocket = request.accept();
  console.log('WebSocket Connection from' + request.remoteAddress + 'accepted.');
  websocket.send ('Hi there. You are now connected to the WebSocket Server');

  websocket.on('message', function(message) {
   if (message.type === 'utf8') {
      console.log('Received Message:' + message.utf8Data);
      websocket.send('Server received your message:'+ message.utf8Data);
  }
});

    websocket.on('close', function(reasonCode, description) {
     console.log('WebSocket Connection from' + request.remoteAddress + 'closed.');
    });
});
```

The first part of the code where we create the HTTP server remains the same. In the newly added code, we start by using the require() method to save a reference to the WebSocket server. We then create a new WebSocketServer object by passing the HTTP server that we created earlier as a configuration option. You can read about the different WebSocketServer configuration options as well as details on the WebSocketServer API at https://github.com/Worlize/WebSocket-Node/wiki/Documentation.

Next we implement the handler for the request event of the server. We first check whether the connection request should be rejected and, if so, call the `reject()` method of the request.

We use a method called `connectionIsAllowed()` to filter connections that need to be rejected. Right now we approve all connections; however, this method can use information such as the connection request's IP address and origin to intelligently filter requests.

If the connection is allowed, we accept the request using the `accept()` method and save the resulting WebSocket connection to the `websocket` variable. This `websocket` variable is the server-side equivalent of the `websocket` variable that we created in the client HTML file earlier.

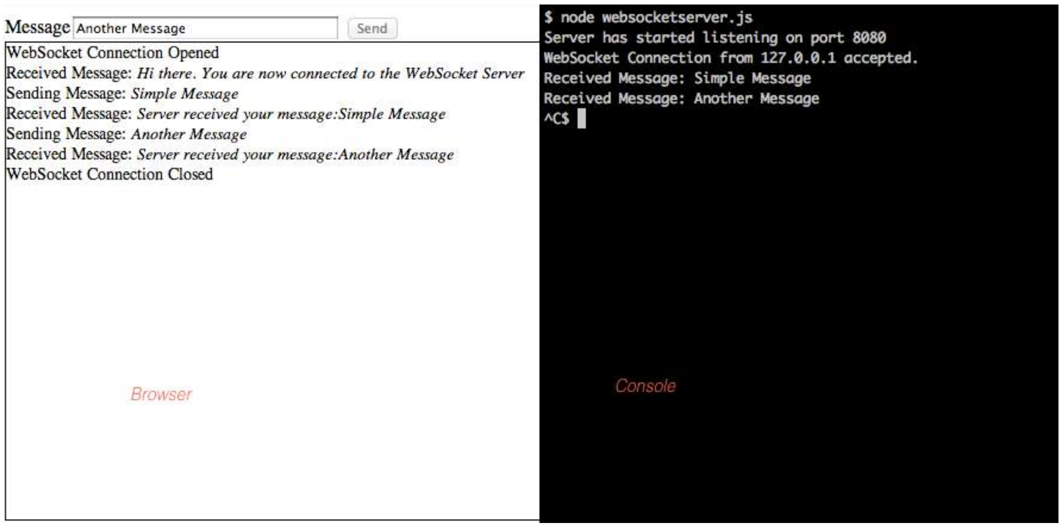
Once we create the connection, we use the `websocket` object's `send()` method to send the client a welcome message notifying it that the connection has been made.

Next we implement the handler for the `message` event of the `websocket` object. Every time a message arrives, we send back a message to the client saying the server just received the message and then log the message to the console.

■ **Note** The WebSocket API allows for multiple message data types such as UTF8 text, binary, and blob data. Unlike on the browser, the `message` object on the server side stores the message data using different properties (such as `utf8Data`, `binaryData`) based on the data type.

Finally, we implement the handler for the `close` event, where we just log the fact that the connection was closed.

If you run the `websocketserver.js` code from the command line and open `websocketclient.html` in the browser, you should see the interaction between the client and the server, as shown in Figure 11-2.



```
WebSocket Connection Opened
Received Message: Hi there. You are now connected to the WebSocket Server
Sending Message: Simple Message
Received Message: Server received your message:Simple Message
Sending Message: Another Message
Received Message: Server received your message:Another Message
WebSocket Connection Closed
```

```
$ node websocketserver.js
Server has started listening on port 8080
WebSocket Connection from 127.0.0.1 accepted.
Received Message: Simple Message
Received Message: Another Message
^C$
```

Figure 11-2. *Interaction between client and server*

As soon as the WebSocket connection is established, the browser receives a welcome message from the server. The Send button also gets enabled on the client. If you type a message and click Send, the server displays the message in the console and sends back a response to the client. Finally, if you shut down the server, the client displays a message that the connection has been closed.

We now have a working example of transmitting plain-text messages back and forth between the client and the server.

■ **Note** It is possible to reduce the message size and optimize bandwidth usage by using binary data instead of plain text. However, we will continue to use UTF8 text even in our game implementation to keep the code simple.

Now that we have looked at the basics of WebSocket communication, it is time to add multiplayer to our game. We will continue from where we left off at the end of Chapter 10.

The first thing we will build is a multiplayer game lobby.

Building the Multiplayer Game Lobby

Our game lobby will display a list of game rooms. Players can join or leave these rooms from the game lobby screen. Once two players join a room, the multiplayer game will start, and the two players can compete with each other.

Defining the Multiplayer Lobby Screen

We will start by adding the HTML code for the multiplayer lobby screen into the `gamecontainer div` in `index.html`, as shown in Listing 11-4.

Listing 11-4. HTML Code for the Multiplayer Lobby Screen (index.html)

```
<div id="multiplayerlobbyscreen" class="gamelayer">
    <select id="multiplayergameslist" size="10">
    </select>
    <input type="button" id="multiplayerjoin" onclick="multiplayer.join();">
    <input type="button" id="multiplayercancel" onclick="multiplayer.cancel();">
</div>
```

The layer contains a `select` element to display the list of game rooms along with two buttons. We will also add the CSS code for the lobby screen to `styles.css`, as shown in Listing 11-5.

Listing 11-5. CSS Code for the Multiplayer Lobby Screen (styles.css)

```
/* Multiplayer Lobby Screen */
#multiplayerlobbyscreen {
    background:url(images/multiplayerlobbyscreen.png) no-repeat center;
}
#multiplayerlobbyscreen input[type="button"]{
    background-image: url(images/buttons.png);
    position:absolute;
    border-width:0px;
    padding:0px;
}
#multiplayerjoin{
    background-position: -2px -212px;
    top:400px;
    left:21px;
    width:74px;
    height:26px;
}
#multiplayerjoin:active,#multiplayerjoin:disabled{
    background-position: -2px -247px;
}
#multiplayercancel{
    background-position: -86px -150px;
    left:545px;
    top:400px;
    width:73px;
    height:24px;
}
#multiplayercancel:active,#multiplayercancel:disabled{
    background-position: -86px -184px;
}
#multiplayergameslist {
    padding:20px;
    position:absolute;
    width:392px;
    height:270px;
    top:98px;
    left:124px;
    background:rgba(0,0,0,0.7);
    border:none;
    color:gray;
    font-size: 15px;
    font-family: 'Courier New', Courier, monospace;
}
#multiplayergameslist:focus {
    outline:none;
}
#multiplayergameslist option.running{
    color:gray;
}
#multiplayergameslist option.waiting{
    color:green;
}
#multiplayergameslist option.empty{
    color:lightblue;
}
```

Now that the lobby screen is in place, we will build the code to connect the browser to the server and populate the games list.

Populating the Games List

We will start by defining a new `multiplayer` object inside `multiplayer.js`, as shown in Listing 11-6.

Listing 11-6. Defining the Multiplayer Object (multiplayer.js)

```
var multiplayer = {
    // Open multiplayer game lobby
    websocket_url:"ws://localhost:8080/",
    websocket:undefined,
    start:function(){
        game.type = "multiplayer";
        var WebSocketObject = window.WebSocket || window.MozWebSocket;
        if (!WebSocketObject){
            game.showMessageBox("Your browser does not support WebSocket. Multiplayer
will not work.");
            return;
        }
        this.websocket = new WebSocketObject(this.websocket_url);
        this.websocket.onmessage = multiplayer.handleWebSocketMessage;
        // Display multiplayer lobby screen after connecting
        this.websocket.onopen = function(){
            // Hide the starting menu layer
            $('.gamelayer').hide();
            $('#multiplayerlobbyscreen').show();
        }
    },
    handleWebSocketMessage:function(message){
        var messageObject = JSON.parse(message.data);
        switch (messageObject.type){
            case "room_list":
                multiplayer.updateRoomStatus(messageObject.status);
                break;
        }
    },
    statusMessages:{
        'starting':'Game Starting',
        'running':'Game in Progress',
        'waiting':'Awaiting second player',
        'empty':'Open'
    },
    updateRoomStatus:function(status){
        var $list = $("#multiplayergameslist");
        $list.empty(); // remove old options
        for (var i=0; i < status.length; i++) {
            var key = "Game "+(i+1)+". "+this.statusMessages[status[i]];
            $list.append($("<option></option>").attr("disabled",status[i]== "running"||status[i]==
"starting").attr("value", (i+1)).text(key).addClass(status[i]).attr("selected",
(i+1)== multiplayer.roomId));
        };
    },
};
```

Inside the multiplayer object, we first define a start() method that tries to initialize a WebSocket connection to the server and saves it in the websocket variable. We then set the websocket object's onmessage event handler to call the handleWebSocketMessage() method and use the onopen event handler to display the lobby screen once the connection is opened.

Next we define the handleWebSocketMessage() method to handle the message data. Instead of passing strings like we did in our WebSocket example earlier, we are going to be passing complete objects between the server and the browser by using JSON.parse() and JSON.stringify() to convert between objects and strings. We start by parsing the message data into a messageObject variable and then use the parsed object's type property to decide how to handle the message.

If the type property is set to room_list, we call the updateRoomStatus() method and pass it the status property.

Finally, we define an updateRoomStatus() method that takes an array of status messages and populates the multiplayergameslist select element. We disable any options that have a status of starting or running and also set the CSS class of the option to the status.

Next, we will need to add a reference to multiplayer.js inside the head section of index.html, as shown in Listing 11-7.

Listing 11-7. Adding Reference to multiplayer.js (index.html)

```
<script src="js/multiplayer.js" type="text/javascript" charset="utf-8"></script>
```

Finally, we will define our multiplayer WebSocket server inside a new file called server.js, as shown in Listing 11-8.

Listing 11-8. Defining the Multiplayer Server (server.js)

```
var WebSocketServer = require('websocket').server;
var http = require('http');

// Create a simple web server that returns the same response for any request
var server = http.createServer(function(request,response){
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.end("This is the node.js HTTP server.");
});

server.listen(8080,function(){
    console.log('Server has started listening on port 8080');
});

var wsServer = new WebSocketServer({
    httpServer:server,
    autoAcceptConnections: false
});

// Logic to determine whether a specified connection is allowed.
function connectionIsAllowed(request){
    // Check criteria such as request.origin, request.remoteAddress
    return true;
}
```

```
// Initialize a set of rooms
var gameRooms = [];
for (var i=0; i < 10; i++) {
    gameRooms.push({status:"empty",players:[],roomId:i+1});
};

var players = [];
wsServer.on('request',function(request){
    if(!connectionIsAllowed(request)){
        request.reject();
        console.log('Connection from' + request.remoteAddress + 'rejected.');
        return;
    }

    var connection = request.accept();
    console.log('Connection from' + request.remoteAddress + 'accepted.');

    // Add the player to the players array
    var player = {
        connection:connection
    }
    players.push(player);

    // Send a fresh game room status list the first time player connects
    sendRoomList(connection);

    // On Message event handler for a connection
    connection.on('message', function(message) {
        if (message.type === 'utf8') {
            var clientMessage = JSON.parse(message.utf8Data);
            switch (clientMessage.type){
                // Handle different message types
            }
        }
    });

    connection.on('close', function(reasonCode, description) {
        console.log('Connection from' + request.remoteAddress + 'disconnected.');
        for (var i = players.length - 1; i >= 0; i--){
            if (players[i]==player){
                players.splice(i,1);
            }
        };
    });
});

function sendRoomList(connection){
    var status = [];
    for (var i=0; i < gameRooms.length; i++) {
        status.push(gameRooms[i].status);
    };
    var clientMessage = {type:"room_list",status:status};
    connection.send(JSON.stringify(clientMessage));
}
```

We start by defining the HTTP server and the WebSocketServer like we did in our earlier `websocketdemo` example.

Next, we define a `rooms` array and fill it with ten `room` objects that have a `status` property set to `empty`.

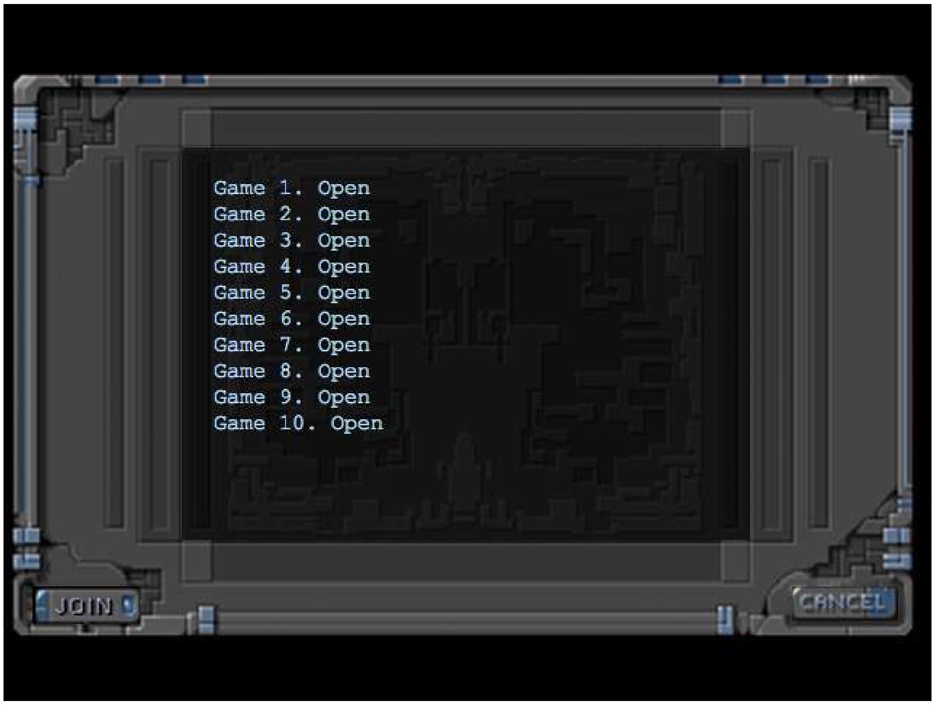
Finally, we implement the connection request event handler. We start by creating a `player` object for the connection and adding it to the `players` array. We then call the `sendRoomList()` method for the connection.

Next, we implement the message event handler for the connection to parse the message data and respond based on the `type` property just like we did on the client. We aren't processing any message types yet.

Next, we implement the `close` event handler where we remove the player from the `players` array once the connection closes.

Finally, we create a `sendRoomList()` method that sends a `status` array inside a `message` object of type `room_list`. This is the same message that we will be parsing on the client side.

If we run the newly created `server.js` and then open our game in the browser, we should be able to click the Multiplayer menu option and arrive at the multiplayer game lobby screen, as shown in Figure 11-3.



```
Game 1. Open
Game 2. Open
Game 3. Open
Game 4. Open
Game 5. Open
Game 6. Open
Game 7. Open
Game 8. Open
Game 9. Open
Game 10. Open
```

Figure 11-3. *The multiplayer game lobby screen*

Behind the scenes, the client is creating a socket connection to the server, and the server is sending back a `room_list` message to the client, which is then used to populate the list.

You should be able to select any of the game rooms but cannot join or leave these rooms. We will now implement joining and leaving a game room.

Joining and Leaving a Game Room

We will start by implementing `join()` and `cancel()` methods inside the `multiplayer` object, as shown in Listing 11-9.

Listing 11-9. Implementing join() and cancel() (multiplayer.js)

```
join:function(){
    var selectedRoom = document.getElementById('multiplayergameslist').value;
    if(selectedRoom){
        multiplayer.sendWebSocketMessage({type:"join_room",roomId:selectedRoom});
        document.getElementById('multiplayergameslist').disabled = true;
        document.getElementById('multiplayerjoin').disabled = true;
    } else {
        game.showMessageBox("Please select a game room to join.");
    }
},
cancel:function(){
    // Leave any existing game room
    if(multiplayer.roomId){
        multiplayer.sendWebSocketMessage({type:"leave_room",roomId:multiplayer.roomId});
        document.getElementById('multiplayergameslist').disabled = false;
        document.getElementById('multiplayerjoin').disabled = false;
        delete multiplayer.roomId;
        delete multiplayer.color;
        return;
    } else {
        // Not in a room, so leave the multiplayer screen itself
        multiplayer.closeAndExit();
    }
},
closeAndExit:function(){
    // clear handlers and close connection
    multiplayer.websocket.onopen = null;
    multiplayer.websocket.onclose = null;
    multiplayer.websocket.onerror = null;
    multiplayer.websocket.close();

    document.getElementById('multiplayergameslist').disabled = false;
    document.getElementById('multiplayerjoin').disabled = false;
    // Show the starting menu layer
    $('.gamelayer').hide();
    $('#gamestartscreen').show();
},
sendWebSocketMessage:function(messageObject){
    this.websocket.send(JSON.stringify(messageObject));
},
```

In the `join()` method, we check whether a room has been selected and, if it has, send a `join_room` WebSocket message to the server with the `roomId` property. We then disable the Join button and the games list. If no room is selected, we ask the player to select a room first.

In the cancel() method, we first check whether the player is in a room using the multiplayer.roomId property. If so, we send a leave_room WebSocket message to the server, delete the roomId and color properties, and enable the Join button and the games list select element. If not, we close the socket connection and return to the game start screen using the closeAndExit() method.

In the closeAndExit() method, we first clear the websocket object's event handlers and close the connection. We then enable the Join button and games list and return to the game start screen.

Finally, we define a sendWebSocketMessage() that converts the messageObject into a string and sends it to the server.

Next, we will modify the server to handle the join_room and leave_room message types by modifying the message event handler in server.js, as shown in Listing 11-10.

Listing 11-10. Handling join_room and leave_room Messages (server.js)

```
// On Message event handler for a connection
connection.on('message', function(message) {
    if (message.type === 'utf8') {
        var clientMessage = JSON.parse(message.utf8Data);
        switch (clientMessage.type){
            case "join_room":
                var room = joinRoom(player,clientMessage.roomId);
                sendRoomListToEveryone();
                break;
            case "leave_room":
                leaveRoom(player,clientMessage.roomId);
                sendRoomListToEveryone();
                break;
        }
    }
});
```

When a join_room message comes in, we first call the joinRoom() method and then send the room list to all players using the sendRoomListToEveryone() method. Similarly, when a leave_room message comes in, we first call the leaveRoom() method and then call the sendRoomListToEveryone() method.

Next, we will define these three new methods inside server.js, as shown in Listing 11-11.

Listing 11-11. The joinRoom(), leaveRoom(), and sendRoomListToEveryone() Methods (server.js)

```
function sendRoomListToEveryone(){
    // Notify all connected players of the room status changes
    var status = [];
    for (var i=0; i < gameRooms.length; i++) {
        status.push(gameRooms[i].status);
    };
    var clientMessage = {type:"room_list",status:status};
    var clientMessageString = JSON.stringify(clientMessage);
    for (var i=0; i < players.length; i++) {
        players[i].connection.send(clientMessageString);
    };
}
```

```
function joinRoom(player,roomId){
    var room = gameRooms[roomId-1];
    console.log("Adding player to room",roomId);
    // Add the player to the room
    room.players.push(player);
    player.room = room;
    // Update room status
    if(room.players.length == 1){
        room.status = "waiting";
        player.color = "blue";
    } else if (room.players.length == 2){
        room.status = "starting";
        player.color = "green";
    }
    // Confirm to player that he was added
    var confirmationMessageString = JSON.stringify({type:"joined_room", roomId:roomId,
color:player.color});
    player.connection.send(confirmationMessageString);
    return room;
}

function leaveRoom(player,roomId){
    var room = gameRooms[roomId-1];
    console.log("Removing player from room",roomId);

    for (var i = room.players.length - 1; i >= 0; i--){
        if(room.players[i]==player){
            room.players.splice(i,1);
        }
    };
    delete player.room;
    // Update room status
    if(room.players.length == 0){
        room.status = "empty";
    } else if (room.players.length == 1){
        room.status = "waiting";
    }
}
```

In the sendRoomListToEveryone() method, we iterate through all the players in the players array and send them a room_list message with the list of rooms.

In the joinRoom() method, we first get the room object using roomId and add the player to the room object's players array. We then set the room's status to waiting or starting depending on how many players are in the room. We also set the player's color to blue or green based on whether the player is the first or second player to join the room. Finally, we send a joined_room message back to the player, with details of the room ID and player color.

In the leaveRoom() method, we first get the room object using roomId and remove the player from the room object's players array. We then set the room object's status to empty or waiting depending on how many players are in the room.

The next change we will make is to handle the joined_room confirmation message inside multiplayer.js, as shown in Listing 11-12.

Listing 11-12. Handling the joined_room Message (multiplayer.js)

```
handleWebSocketMessage:function(message){
    var messageObject = JSON.parse(message.data);
    switch (messageObject.type){
        case "room_list":
            multiplayer.updateRoomStatus(messageObject.status);
            break;
        case "joined_room":
            multiplayer.roomId = messageObject.roomId;
            multiplayer.color = messageObject.color;
            break;
    }
},
```

When a joined_room message comes in, we save the roomId and color properties inside the multiplayer object.
Finally, we will ensure that a player is removed from a game room if the player is disconnected by modifying the close event handler on the server, as shown in Listing 11-13.

Listing 11-13. Handling Player Disconnects (server.js)

```
connection.on('close', function(reasonCode, description) {
    console.log('Connection from' + request.remoteAddress + 'disconnected.');

    for (var i = players.length - 1; i >= 0; i--){
        if (players[i]==player){
            players.splice(i,1);
        }
    };

    // If the player is in a room, remove him from room and notify everyone
    if(player.room){
        var status = player.room.status;
        var roomId = player.room.roomId;
        // If the game was running, end the game as well
        leaveRoom(player,roomId);
        sendRoomListToEveryone();
    }
});
```

In the newly added code, we check whether the disconnected player is in a room, and if so, we remove the player from the room using the leaveRoom() method and then notify everyone using the sendRoomListToEveryone() method.
If we restart the server and run the game in more than one browser window, we should be able to join a room in one window and see the status change in both the windows, as shown in Figure 11-4.

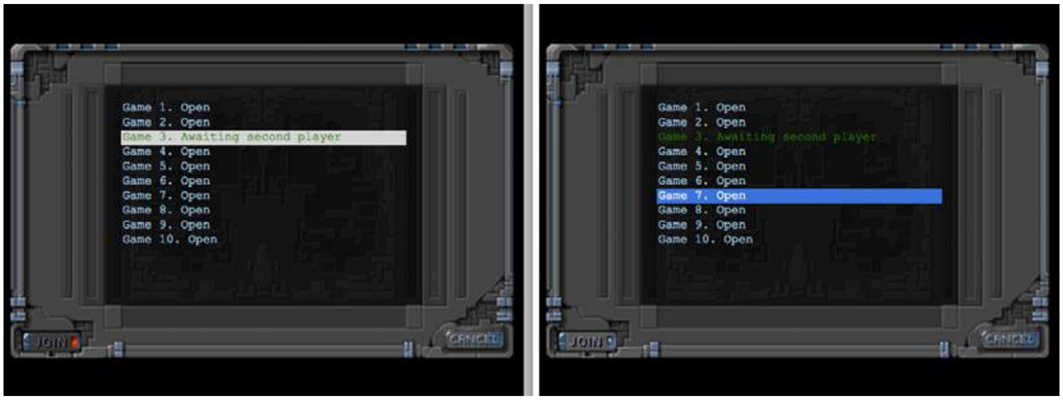


Figure 11-4. *Room status updated on both browsers when joining a room*

You will notice that the Join button and the list get disabled once you join a room. If you join the same room on both browsers, the room status changes to `starting` and no one else can join the room.

If you click Cancel, you will leave the room, and the Join button will be reenabled. If you click Cancel again, you are taken back to the main menu. If you disconnect from the server by closing the browser window, you will be removed from the room.

We now have a working game lobby where players can join and leave game rooms. Next we will start the multiplayer game once the players join the game room.

Starting the Multiplayer Game

Our multiplayer game will start once two players join a game room. We will need to tell both the clients to load the same level. Once the level has loaded on both browsers, we will then start the game. The first thing we need to do is define a new multiplayer level.

Defining the Multiplayer Level

The multiplayer level, while being similar to the single-player levels, will contain some extra information such as the starting location for each player and the starting items for each team. We will start by defining a simple level inside a new `multiplayer` array in the `maps` object, as shown in Listing 11-14.

Listing 11-14. A Multiplayer Level Inside the Multiplayer Array (maps.js)

```
"multiplayer":[
    {
        /* Map Details */
        "mapImage":"images/maps/level-one.png",

        /* Map coordinates that are obstructed by terrain*/
        "mapGridWidth":60,
        "mapGridHeight":40,
        "mapObstructedTerrain":[
            [49,8], [50,8], [51,8], [51,9], [52,9], [53,9], [53,10], [53,11], [53,12], [53,13],
[53,14], [53,15], [53,16], [52,16], [52,17], [52,18], [52,19], [51,19], [50,19], [50,18], [50,17],
[49,17], [49,18], [48,18], [47,18], [47,17], [47,16], [48,16], [49,16], [49,15], [49,14], [48,14],
[48,13], [48,12], [49,12], [49,11], [50,11], [50,10], [49,10], [49,9], [44,0], [45,0], [45,1],
[45,2], [46,2], [47,2], [47,3], [48,3], [48,4], [48,5], [49,5], [49,6], [49,7], [50,7], [51,7],
[51,6], [51,5], [51,4], [52,4], [53,4], [53,3], [54,3], [55,3], [55,2], [56,2], [56,1], [56,0],
[55,0], [43,19], [44,19], [45,19], [46,19], [47,19], [48,19], [48,20], [48,21], [47,21], [46,21],
[45,21], [44,21], [43,21], [43,20], [41,22], [42,22], [43,22], [44,22], [45,22], [46,22], [47,22],
[48,22], [49,22], [50,22], [50,23], [50,24], [49,24], [48,24], [47,24], [47,25], [47,26], [47,27],
[47,28], [47,29], [47,30], [46,30], [45,30], [44,30], [43,30], [43,29], [43,28], [43,27], [43,26],
[43,25], [43,24], [42,24], [41,24], [41,23], [48,39], [49,39], [50,39], [51,39], [52,39], [53,39],
[54,39], [55,39], [56,39], [57,39], [58,39], [59,39], [59,38], [59,37], [59,36], [59,35], [59,34],
[59,33], [59,32], [59,31], [59,30], [59,29], [0,0], [1,0], [2,0], [1,1], [2,1], [10,3], [11,3],
[12,3], [12,2], [13,2], [14,2], [14,3], [14,4], [15,4], [15,5], [15,6], [14,6], [13,6], [13,5],
[12,5], [11,5], [10,5], [10,4], [3,9], [4,9], [5,9], [5,10], [6,10], [7,10], [8,10], [9,10], [9,11],
[10,11], [11,11], [11,10], [12,10], [13,10], [13,11], [13,12], [12,12], [11,12], [10,12], [9,12],
[8,12], [7,12], [7,13], [7,14], [6,14], [5,14], [5,13], [5,12], [5,11], [4,11], [3,11], [3,10],
[33,33], [34,33], [35,33], [35,34], [35,35], [34,35], [33,35], [33,34], [27,39], [27,38], [27,37],
[28,37], [28,36], [28,35], [28,34], [28,33], [28,32], [28,31], [28,30], [28,29], [29,29], [29,28],
[29,27], [29,26], [29,25], [29,24], [29,23], [30,23], [31,23], [32,23], [32,22], [32,21], [31,21],
[30,21], [30,22], [29,22], [28,22], [27,22], [26,22], [26,21], [25,21], [24,21], [24,22], [24,23],
[25,23], [26,23], [26,24], [25,24], [25,25], [24,25], [24,26], [24,27], [25,27], [25,28], [25,29],
[24,29], [23,29], [23,30], [23,31], [24,31], [25,31], [25,32], [25,33], [24,33], [23,33], [23,34],
[23,35], [24,35], [24,36], [24,37], [23,37], [22,37], [22,38], [22,39], [23,39], [24,39], [25,39],
[26,0], [26,1], [25,1], [25,2], [25,3], [26,3], [27,3], [27,2], [28,2], [29,2], [29,3], [30,3],
[31,3], [31,2], [31,1], [32,1], [32,0], [33,0], [32,8], [33,8], [34,8], [34,9], [34,10], [33,10],
[32,10], [32,9], [8,29], [9,29], [9,30], [17,32], [18,32], [19,32], [19,33], [18,33], [17,33],
[18,34], [19,34], [3,27], [4,27], [4,26], [3,26], [2,26], [3,25], [4,25], [9,20], [10,20], [11,20],
[11,21], [10,21], [10,19], [19,7], [15,7], [29,12], [30,13], [20,14], [21,14], [34,13], [35,13],
[36,13], [36,14], [35,14], [34,14], [35,15], [36,15], [16,18], [17,18], [18,18], [16,19], [17,19],
[18,19], [17,20], [18,20], [11,19], [58,0], [59,0], [58,1], [59,1], [59,2], [58,3], [59,3], [58,4],
[59,4], [59,5], [58,6], [59,6], [58,7], [59,7], [59,8], [58,9], [59,9], [58,10], [59,10], [59,11],
[52,6], [53,6], [54,6], [52,7], [53,7], [54,7], [53,8], [54,8], [44,17], [46,32], [55,32], [54,28],
[26,34], [34,34], [4,10], [6,11], [6,12], [6,13], [7,11], [8,11], [12,11], [27,0], [27,1], [26,2],
[28,1], [28,0], [29,0], [29,1], [30,2], [30,1], [30,0], [31,0], [33,9], [46,0], [47,0], [48,0],
[49,0], [50,0], [51,0], [52,0], [53,0], [54,0], [55,1], [54,1], [53,1], [52,1], [51,1], [50,1],
[49,1], [48,1], [47,1], [46,1], [48,2], [49,2], [50,2], [51,2], [52,2], [53,2], [54,2], [52,3],
[51,3], [50,3], [49,3], [49,4], [50,4], [50,5], [50,6], [50,9], [51,10], [52,10], [51,11], [52,11],
[50,12], [51,12], [52,12], [49,13], [50,13], [51,13], [52,13], [50,14], [51,14], [52,14], [50,15],
[51,15], [52,15], [50,16], [51,16], [51,17], [48,17], [51,18], [44,20], [45,20], [46,20], [47,20],
[42,23], [43,23], [44,23], [45,23], [46,23], [47,23], [48,23], [49,23], [44,24], [45,24], [46,24],
[44,25], [45,25], [46,25], [44,26], [45,26], [46,26], [44,27], [45,27], [46,27], [44,28], [45,28],
[46,28], [44,29], [45,29], [46,29], [11,4], [12,4], [13,4], [13,3], [14,5], [25,22], [31,22],
[27,23], [28,23], [27,24], [28,24], [26,25], [27,25], [28,25], [25,26], [26,26], [27,26], [28,26],
[26,27], [27,27], [28,27], [26,28], [27,28], [28,28], [26,29], [27,29], [24,30], [25,30], [26,30],
[27,30], [26,31], [27,31], [26,32], [27,32], [26,33], [27,33], [24,34], [25,34], [27,34], [25,35],
[26,35], [27,35], [25,36], [26,36], [27,36], [25,37], [26,37], [23,38], [24,38], [25,38], [26,38],
[26,39], [2,25], [9,19], [36,31]
        ],

        /* Entities to be loaded */
        "requirements":{
            "buildings":["base","harvester","starport","ground-turret"],
            "vehicles":["transport","scout-tank","heavy-tank","harvester"],
            "aircraft":["wraith","chopper"],
            "terrain":["oilfield"]
        },

        /* Economy Related*/
        "cash":{
            "blue":1000,
            "green":1000
        },

        /* Entities to be added */
        "items":[
            {"type":"terrain","name":"oilfield","x":16,"y":4,"action":"hint"},
            {"type":"terrain","name":"oilfield","x":34,"y":12,"action":"hint"},
            {"type":"terrain","name":"oilfield","x":1,"y":30,"action":"hint"},
            {"type":"terrain","name":"oilfield","x":38,"y":38,"action":"hint"},
        ],

        /* Entities for each starting team */
        "teamStartingItems":[
            {"type":"buildings","name":"base","x":0,"y":0},
            {"type":"vehicles","name":"harvester","x":2,"y":0},
            {"type":"vehicles","name":"heavy-tank","x":2,"y":1},
            {"type":"vehicles","name":"scout-tank","x":3,"y":0},
            {"type":"vehicles","name":"scout-tank","x":3,"y":1},
        ],
        "spawnLocations":[
            { "x":48, "y":36,"startX":36,"startY":20},
            { "x":3, "y":36,"startX":0,"startY":20},
            { "x":36, "y":3,"startX":32,"startY":0},
            { "x":3, "y":3,"startX":0,"startY":0},
        ],
        /* Conditional and Timed Trigger Events */
        "triggers":[
        ]
    }
]
```

The two new elements that we have introduced in the multiplayer level are the `teamStartingItems` and `spawnLocations` arrays.

The `teamStartingItems` array contains a list of items that each team will have at the beginning of the level. The x and y coordinates will be relative to the location where the team is spawned.

The `spawnLocations` array contains a few spots on the map where each player team can start. Each object contains the x and y coordinates of the location, as well as the starting panning offset for the location.

Now that we have defined the multiplayer level, we need to load the level once the two players join a game room.

Loading the Multiplayer Level

When two players join a room, we will tell them both to initialize the level and wait for both to confirm that the level has been initialized. Once this happens, we need to tell them both to start the game.

We will start by adding a few new methods to `server.js` to handle initializing and starting the game, as shown in Listing 11-15.

Listing 11-15. Initializing and Starting the Game (server.js)

```
function initGame(room){
    console.log("Both players Joined. Initializing game for Room "+room.roomId);

    // Number of players who have loaded the level
    room.playersReady = 0;

    // Load the first multiplayer level for both players
    // This logic can change later to let the players pick a level
    var currentLevel = 0;

    // Randomly select two spawn locations between 0 and 3 for both players.
    var spawns = [0,1,2,3];
    var spawnLocations = {"blue":spawns.splice(Math.floor(Math.random()*spawns.length),1),
"green":spawns.splice(Math.floor(Math.random()*spawns.length),1)};

    sendRoomWebSocketMessage(room,{type:"init_level", spawnLocations:spawnLocations,
level:currentLevel});
}

function startGame(room){
    console.log("Both players are ready. Starting game in room",room.roomId);
    room.status = "running";
    sendRoomListToEveryone();
    // Notify players to start the game
    sendRoomWebSocketMessage(room,{type:"start_game"});
}

function sendRoomWebSocketMessage(room,messageObject){
    var messageString = JSON.stringify(messageObject);
    for (var i = room.players.length - 1; i >= 0; i--){
        room.players[i].connection.send(messageString);
    };
}
```

In the `initGame()` method, we initialize the `playersReady` variable, select two random spawn locations for both the players, and send the location to both the players inside an `init_level` message using the `sendRoomWebSocketMessage()` method.

In the `startGame()` method, we set the room status to `running`, update every player's room list, and finally send both players the `start_game` message using the `sendRoomWebSocketMessage()` method.

Finally, in the `sendRoomWebSocketMessage()` method we iterate through the players in a room and send each of them a given message.

Next, we will modify the message event handler in `server.js` to initialize and start the game, as shown in Listing 11-16.

Listing 11-16. Modifying the Message Event Handler (server.js)

```
// On Message event handler for a connection
connection.on('message', function(message) {
    if (message.type === 'utf8') {
        var clientMessage = JSON.parse(message.utf8Data);
        switch (clientMessage.type){
            case "join_room":
                var room = joinRoom(player,clientMessage.roomId);
                sendRoomListToEveryone();
                if(room.players.length == 2){
                    initGame(room);
                }
                break;
            case "leave_room":
                leaveRoom(player,clientMessage.roomId);
                sendRoomListToEveryone();
                break;
            case "initialized_level":
                player.room.playersReady++;
                if (player.room.playersReady==2){
                    startGame(player.room);
                }
                break;
        }
    }
});
```

When a player joins a room and the player count reaches two, we call the `initGame()` method. When we receive an `initialized_level` message from a player, we increment the `playersReady` variable. Once the count reaches two, we call the `startGame()` method.

Next we will add two new methods to the `multiplayer` object to initialize the multiplayer level and start the game, as shown in Listing 11-17.

Listing 11-17. Initializing and Starting the Multiplayer Game (multiplayer.js)

```
currentLevel:0,
initMultiplayerLevel:function(messageObject){
    $('.gamelayer').hide();
    var spawnLocations = messageObject.spawnLocations;

    // Initialize multiplayer related variables
    multiplayer.commands = [[]];
    multiplayer.lastReceivedTick = 0;
    multiplayer.currentTick = 0;

    game.team = multiplayer.color;

    // Load all the items for the level
    multiplayer.currentLevel = messageObject.level;
    var level = maps.multiplayer[multiplayer.currentLevel];

    // Load all the assets for the level
    game.currentMapImage = loader.loadImage(level.mapImage);
    game.currentLevel = level;

    // Setup offset based on spawn location sent by server
    // Load level Requirements
    game.resetArrays();
    for (var type in level.requirements){
           var requirementArray = level.requirements[type];
           for (var i=0; i < requirementArray.length; i++) {
               var name = requirementArray[i];
               if (window[type]){
                   window[type].load(name);
               } else {
                   console.log('Could not load type :',type);
               }
           };
     }

    for (var i = level.items.length - 1; i >= 0; i--){
        var itemDetails = level.items[i];
        game.add(itemDetails);
    };

    // Add starting items for both teams at their respective spawn locations
    for (team in spawnLocations){
        var spawnIndex = spawnLocations[team];
        for (var i=0; i < level.teamStartingItems.length; i++) {
            var itemDetails = $.extend(true,{},level.teamStartingItems[i]);
            itemDetails.x += level.spawnLocations[spawnIndex].x+itemDetails.x;
            itemDetails.y += level.spawnLocations[spawnIndex].y+itemDetails.y;
            itemDetails.team = team;
            game.add(itemDetails);
        };

        if (team==game.team){
            game.offsetX = level.spawnLocations[spawnIndex].startX*game.gridSize;
            game.offsetY = level.spawnLocations[spawnIndex].startY*game.gridSize;
        }
    }


    // Create a grid that stores all obstructed tiles as 1 and unobstructed as 0
    game.currentMapTerrainGrid = [];
    for (var y=0; y < level.mapGridHeight; y++) {
        game.currentMapTerrainGrid[y] = [];
           for (var x=0; x< level.mapGridWidth; x++) {
            game.currentMapTerrainGrid[y][x] = 0;
        }
    };
    for (var i = level.mapObstructedTerrain.length - 1; i >= 0; i--){
        var obstruction = level.mapObstructedTerrain[i];
        game.currentMapTerrainGrid[obstruction[1]][obstruction[0]] = 1;
    };
    game.currentMapPassableGrid = undefined;
```

```
    // Load Starting Cash For Game
    game.cash = $.extend([],level.cash);

    // Enable the enter mission button once all assets are loaded
    if (loader.loaded){
        multiplayer.sendWebSocketMessage({type:"initialized_level"});

    } else {
        loader.onload = function(){
            multiplayer.sendWebSocketMessage({type:"initialized_level"});
        }
    }
},
startGame:function(){
    fog.initLevel();
    game.animationLoop();
    game.start();
},
```

In the `initMultiplayerLevel()` method, we start by initializing a few multiplayer-related variables that we will need later. We then initialize the `game.team` and `multiplayer.currentLevel` variables. We then load the multiplayer map like we did for the single-player campaign.

Next, we place all the starting items for both the players at their respective spawn locations and set the offset location for each player based on their spawn locations.

We then load the terrain grid like we did for single player, and finally we send the `initialized_level` message back to the server once the map has loaded completely to let the server know that the client has finished loading the level.

In the `startGame()` method, we initialize the fog, call the `animationLoop()` once, and finally call `game.start()`.

Next, we will modify the `handleWebSocketMessage()` method in `multiplayer.js` to call these newly created methods, as shown in Listing 11-18.

Listing 11-18. Modifying Message Handler to Initialize and Start Game (multiplayer.js)

```
handleWebSocketMessage:function(message){
    var messageObject = JSON.parse(message.data);
    switch (messageObject.type){
        case "room_list":
            multiplayer.updateRoomStatus(messageObject.status);
            break;
        case "joined_room":
            multiplayer.roomId = messageObject.roomId;
            multiplayer.color = messageObject.color;
            break;
        case "init_level":
            multiplayer.initMultiplayerLevel(messageObject);
            break;
        case "start_game":
            multiplayer.startGame();
            break;
    }
},
```

We merely call the `initMultiplayerLevel()` method when we receive an `init_level` message and the `startGame()` method when we receive a `start_game` message.

If we restart the server and run the game in two browser windows, we should be able to join the same room from both browsers and see the game load in both, as shown in Figure 11-5.

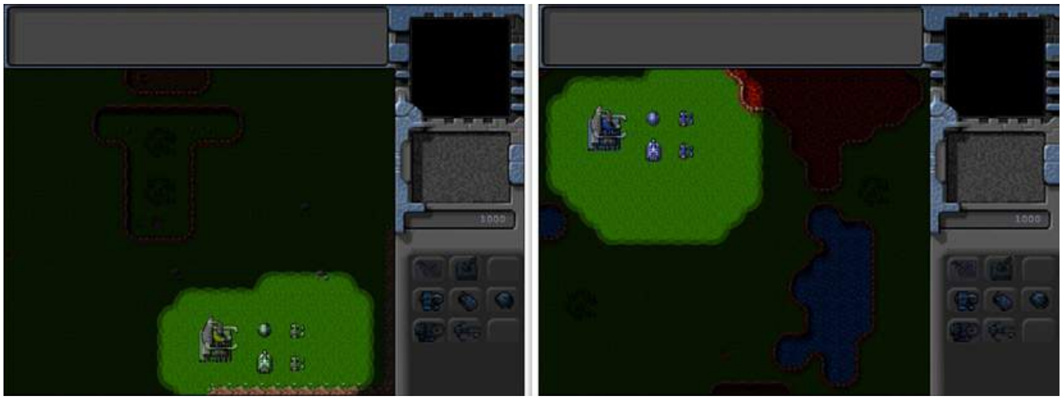


Figure 11-5. *Multiplayer game loading in both browser windows*

Once the two players join the room, the server automatically assigns both the players different colors and spawn locations. When the game loads, both players are placed at their respective spawn locations with the same starting team: two scout tanks, a heavy tank, and a harvester.

We can scroll around the map and even select units; however, we still can't play the game by giving these units commands. This is what we will implement in the next chapter.

Summary

In this chapter, we looked at using the WebSocket API with Node.js for a simple client-server architecture. First we installed Node.js and the WebSocket-Node package and used it to build a simple WebSocket server. Then we built a simple WebSocket-based browser client and sent messages back and forth between the browser and the server.

We used this same architecture to implement a multiplayer game lobby with rooms that players could join and leave. We designed a multiplayer level with spawn locations and starting teams. Finally, we loaded and started the same level on two different browsers, while placing the two players at different spawn locations.

In the next chapter, we will implement the actual multiplayer gameplay by passing commands between the browsers and server. We will use triggers to implement winning and losing a game. Finally, we will add some finishing touches and wrap up the multiplayer section of our game.