$ 29.99 US
£ 19.99 UK

Prices do not include local sales tax or VAT

What you will learn from this book

- Design games using various multiplayer models
- Create authoritative game servers
- Develop game clients that communicate and synchronize with other clients through WebSockets
- Write responsive multiplayer games using client prediction
- Upgrade an existing single-player game by adding multiplayer mechanics
- Use state-of-the-art technologies to make the development process and its end product enjoyable to programmers and amazing to the player
- Create peer-to-peer gaming using WebRTC

Who this book is written for

If you are a HTML5 game developer who can make basic single-player games and you are now ready to incorporate multiplayer functionality in your games as quickly as possible, then this book is ideal for you.

Developing an online game can be just as much fun as playing it. However, orchestrating multiple clients and keeping everyone in sync with a game server, reducing and managing network latency (all the while preventing cheating), and making sure every player has an excellent experience can quickly become overwhelming.

This book will teach you how to develop games that support multiple players interacting in the same game world, and show you how to perform network programming operations in order to implement such systems. It covers the fundamentals of game networking by developing a real-time multiplayer game of Tic-tac-toe before moving on to convert an existing 2D single-player snake game to multiplayer, using a more scalable game design for online gaming.

Finally you will be tackling more advanced networking topics, allowing you to handle problems such as server queries from multiple users and making your multiplayer games more secure and less prone to cheating.

Multiplayer Game Development with HTML5

Build fully-featured, highly interactive multiplayer games with HTML5

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Getting Started with Multiplayer Game Programming'
- A synopsis of the book’s content
- More information on Multiplayer Game Development with HTML5

About the Author

Rodrigo Silveira is a software engineer at Deseret Digital Media. There, he divides his time developing applications in PHP, JavaScript, and Java for Android. Some of his hobbies outside of work include blogging and recording educational videos about software development, learning about new technologies, and finding ways to push the Web forward.

Rodrigo received his bachelor’s degree in computer science from Brigham Young University, Idaho, as well as an associate's degree in business management from LDS Business College in Salt Lake City, Utah.

His fascination for game development began in his early teenage years, and his skills grew as he discovered the power of a library subscription, a curious and willing mind, and supportive parents and friends.

Today, Rodrigo balances his time between the three great passions of his life—his family, software development, and video games (with the last two usually being mingled together).
Multiplayer Game Development with HTML5

Welcome to Multiplayer Game Development with HTML5. This book will teach you how to develop games that support interacting multiple players in the same game world, and how to perform network programming operations in order to implement such systems. It covers topics such as WebSockets and client-side and server-side game programming in JavaScript with Node.js, latency reduction techniques, and handling server queries from multiple users. We will accomplish this by walking you through the process of developing two actual multiplayer games from start to finish, and it will also teach you about various topics in HTML5 game development in the process. The aim of the book is to teach you to create game worlds for multiple players who want to compete or interact through the Internet using HTML5.

What This Book Covers

Chapter 1, Getting Started with Multiplayer Game Programming, introduces network programming, with emphasis on designing a multiplayer game. It illustrates the basic concepts of multiplayer game development by walking you through the creation of a real-time game of Tic Tac Toe.

Chapter 2, Setting Up the Environment, describes the current state of the art in the JavaScript development world, including JavaScript in the server through Node.js. It also describes the current techniques to manage the development cycle in JavaScript with workflow and resource management tools, such as Npm, Bower, Grunt, and more.

Chapter 3, Feeding a Snake in Real Time, takes an existing single-player Snake game and builds in the ability to play with multiple players in the same game world using the tools heretofore described. Concepts of lobby, rooms, matchmaking, and handling queries from users are also described and demonstrated, which add the functionality to the Snake game. The chapter introduces the most powerful and widely used WebSocket abstraction in the industry today—socket.io.

Chapter 4, Reducing Network Latency, teaches you techniques to reduce network latency in order to create a smooth playing experience. The most common among such techniques—client prediction—is demonstrated and incorporated into the Snake game that is described in the previous chapter. The game server code is also updated for the purpose of performance by introducing a second update loop.

Chapter 5, Leveraging the Bleeding Edge, describes the exciting opportunities that are found in game development on the web platform. It explains WebRTC, HTML5's gamepad, the fullscreen mode, and media capture APIs. Other promised and experimental technologies and APIs are also described here.
Chapter 6, Adding Security and Fair Play, covers common flaws and security vulnerabilities that are associated with network gaming. Here, common techniques are described and demonstrated, allowing you to develop games that provide a playing experience that is free from cheating.
Getting Started with Multiplayer Game Programming

If you're reading this book, chances are pretty good that you are already a game developer. That being the case, then you already know just how exciting it is to program your own games, either professionally or as a highly gratifying hobby that is very time-consuming. Now you're ready to take your game programming skills to the next level—that is, you're ready to implement multiplayer functionality into your JavaScript-based games.

In case you have already set out to create multiplayer games for the Open Web Platform using HTML5 and JavaScript, then you may have already come to realize that a personal desktop computer, laptop, or a mobile device is not particularly the most appropriate device to share with another human player for games in which two or more players share the same game world at the same time. Therefore, what is needed in order to create exciting multiplayer games with JavaScript is some form of networking technology.

In this chapter, we will discuss the following principles and concepts:

- The basics of networking and network programming paradigms
- Socket programming with HTML5
- Programming a game server and game clients
- Turn-based multiplayer games
Understanding the basics of networking

It is said that one cannot program games that make use of networking without first understanding all about the discipline of computer networking and network programming. Although having a deep understanding of any topic can be only beneficial to the person working on that topic, I don’t believe that you must know everything there is to know about game networking in order to program some pretty fun and engaging multiplayer games. Saying that is the case is like saying that one needs to be a scholar of the Spanish language in order to cook a simple burrito. Thus, let us take a look at the most basic and fundamental concepts of networking. At the end of this section, you will know enough about computer networking to get started, and you will feel comfortable adding multiplayer aspects to your games.

One thing to keep in mind is that, even though networked games are not nearly as old as single-player games, computer networking is actually a very old and well-studied subject. Some of the earliest computer network systems date back to the 1950s. Though some of the techniques have improved over the years, the basic idea remains the same: two or more computers are connected together to establish communication between the machines. By communication, I mean data exchange, such as sending messages back and forth between the machines, or one of the machines only sends the data and the other only receives it.

With this brief introduction to the concept of networking, you are now grounded in the subject of networking, enough to know what is required to network your games—two or more computers that talk to each other as close to real time as possible.

By now, it should be clear how this simple concept makes it possible for us to connect multiple players into the same game world. In essence, we need a way to share the global game data among all the players who are connected to the game session, then continue to update each player about every other player. There are several different techniques that are commonly used to achieve this, but the two most common approaches are peer-to-peer and client-server. Both techniques present different opportunities, including advantages and disadvantages. In general, neither is particularly better than the other, but different situations and use cases may be better suited for one or the other technique.

Peer-to-peer networking

A simple way to connect players into the same virtual game world is through the peer-to-peer architecture. Although the name might suggest that only two peers ("nodes") are involved, by definition a peer-to-peer network system is one in which two or more nodes are connected directly to each other without a centralized system orchestrating the connection or information exchange.
On a typical peer-to-peer setup, each peer serves the same function as every other one—that is, they all consume the same data and share whatever data they produce so that others can stay synchronized. In the case of a peer-to-peer game, we can illustrate this architecture with a simple game of Tic-tac-toe.

Once both the players have established a connection between themselves, whoever is starting the game makes a move by marking a cell on the game board. This information is relayed across the wire to the other peer, who is now aware of the decision made by his or her opponent, and can thus update their own game world. Once the second player receives the game's latest state that results from the first player's latest move, the second player is able to make a move of their own by checking some available space on the board. This information is then copied over to the first player who can update their own world and continue the process by making the next desired move.

The process goes on until one of the peers disconnects or the game ends as some condition that is based on the game's own business logic is met. In the case of the game of Tic-tac-toe, the game would end once one of the players has marked three spaces on the board forming a straight line or if all nine cells are filled, but neither player managed to connect three cells in a straight path.
Some of the benefits of peer-to-peer networked games are as follows:

- **Fast data transmission**: Here, the data goes directly to its intended target. In other architectures, the data could go to some centralized node first, then the central node (or the "server", as we'll see in the next section) contacts the other peer, sending the necessary updates.

- **Simpler setup**: You would only need to think about one instance of your game that, generally speaking, handles its own input, sends its input to other connected peers, and handles their output as input for its own system. This can be especially handy in turn-based games, for example, most board games such as Tic-tac-toe.

- **More reliability**: Here one peer that goes offline typically won't affect any of the other peers. However, in the simple case of a two-player game, if one of the players is unable to continue, the game will likely cease to be playable. Imagine, though, that the game in question has dozens or hundreds of connected peers. If a handful of them suddenly lose their Internet connection, the others can continue to play. However, if there is a server that is connecting all the nodes and the server goes down, then none of the other players will know how to talk to each other, and nobody will know what is going on.

On the other hand, some of the more obvious drawbacks of peer-to-peer architecture are as follows:

- **Incoming data cannot be trusted**: Here, you don't know for sure whether or not the sender modified the data. The data that is input into a game server will also suffer from the same challenge, but once the data is validated and broadcasted to all the other peers, you can be more confident that the data received by each peer from the server will have at least been sanitized and verified, and will be more credible.

- **Fault tolerance can be very low**: The opposite argument was made in the benefits' section of Peer-to-peer networking that we discussed previously; if enough players share the game world, one or more crashes won't make the game unplayable to the rest of the peers. Now, if we consider the many cases where any of the players that suddenly crash out of the game negatively affect the rest of the players, we can see how a server could easily recover from the crash.
• **Data duplication when broadcasting to other peers:** Imagine that your game is a simple 2D side scroller, and many other players are sharing that game world with you. Every time one of the players moves to the right, you receive the new (x, y) coordinates from that player, and you're able to update your own game world. Now, imagine that you move your player to the right by a very few pixels; you would have to send that data out to all of the other nodes in the system.

Overall, peer-to-peer is a very powerful networking architecture and is still widely used by many games in the industry. Since current peer-to-peer web technologies are still in their infancy, most JavaScript-powered games today do not make use of peer-to-peer networking. For this and other reasons that should become apparent soon, we will focus the rest of the book almost exclusively on the other popular networking paradigm, namely, the client-server architecture.

### Client-server networking

The idea behind the **client-server networking** architecture is very simple. If you squint your eyes hard enough, you can almost see a peer-to-peer graph. The most obvious difference between them, is that, instead of every node being an equal peer, one of the nodes is special. That is, instead of every node connecting to every other node, every node (client) connects to a main centralized node called the **server**.

While the concept of a client-server network seems clear enough, perhaps a simple metaphor might make it easier for you to understand the role of each type of node in this network format as well as differentiate it from peer-to-peer (McConnell, Steve, (2004) *Code Complete*, Microsoft Press). In a peer-to-peer network, you can think of it as a group of friends (peers) having a conversation at a party. They all have access to all the other peers involved in the conversation and can talk to them directly. On the other hand, a client-server network can be viewed as a group of friends having dinner at a restaurant. If a client of the restaurant wishes to order a certain item from the menu, he or she must talk to the waiter, who is the only person in that group of people with access to the desired products and the ability to serve the products to the clients.
In short, the server is in charge of providing data and services to one or more clients. In the context of game development, the most common scenario is when two or more clients connect to the same server; the server will keep track of the game as well as the distributed players. Thus, if two players are to exchange information that is only pertinent to the two of them, the communication will go from the first player to and through the server and will end up at the other end with the second player.

Following the example of the two players involved in a game of *Tic-tac-toe* that we looked at in the section about peer-to-peer, we can see how similar the flow of events is on a client-server model. Again, the main difference is that players are unaware of each other and only know what the server tells them.

While you can very easily mimic a peer-to-peer model by using a server to merely connect the two players, most often the server is used much more actively than that. There are two ways to engage the server in a networked game, namely in an authoritative and a non-authoritative way. That is to say, you can have the enforcement of the game's logic strictly in the server, or you can have the clients handle the game logic, input validation, and so on. Today, most games using the client-server architecture actually use a hybrid of the two (authoritative and non-authoritative servers, which we'll discuss later in the book). For all intents and purposes, however, the server's purpose in life is to receive input from each of the clients and distribute that input throughout the pool of connected clients.

Now, regardless of whether you decide to go with an authoritative server instead of a non-authoritative one, you will notice that one of challenges with a client-server game is that you will need to program both ends of the stack. You will have to do this even if your clients do nothing more than take input from the user, forward it to the server, and render whatever data they receive from the server; if your game server does nothing more than forward the input that it receives from each client to every other client, you will still need to write a game client and a game server.

We will discuss game clients and servers later in the chapter. For now, all we really need to know is that these two components are what set this networking model apart from peer-to-peer.
Some of the benefits of client-server networked games are as follows:

- **Separation of concerns**: If you know anything about software development, you know that this is something you should always aim for. That is, good, maintainable software is written as discrete components where each does one "thing", and it is done well. Writing individual specialized components lets you focus on performing one individual task at a time, making your game easier to design, code, test, reason, and maintain.

- **Centralization**: While this can be argued against as well as in favor of, having one central place through which all communication must flow makes it easier to manage such communication, enforce any required rules, control access, and so forth.

- **Less work for the client**: Instead of having a client (peer) in charge of taking input from the user as well as other peers, validating all the input, sharing data among other peers, rendering the game, and so on, the client can focus on only doing a few of these things, allowing the server to offload some of this work. This is particularly handy when we talk about mobile gaming, and how much subtle divisions of labor can impact the overall player experience. For example, imagine a game where 10 players are engaged in the same game world. In a peer-to-peer setup, every time one player takes an action, he or she would need to send that action to nine other players (in other words, there would need to be nine network calls, boiling down to more mobile data usage). On the other hand, on a client-server configuration, one player would only need to send his or her action to one of the peers, that is, the server, who would then be responsible for sending that data to the remaining nine players.

Common drawbacks of client-server architectures, whether or not the server is authoritative, are as follows:

- **Communication takes longer to propagate**: In the very best possible scenario imaginable, every message sent from the first player to the second player would take twice as long to be delivered as compared to a peer-to-peer connection. That is, the message would be first sent from the first player to the server and then from the server to the second player. There are many techniques that are used today to solve the latency problem faced in this scenario, some of which we will discuss in much more depth later in Chapter 4, Reducing Network Latency. However, the underlying dilemma will always be there.
• **More complexity due to more moving parts:** It doesn't really matter how you slice the pizza; the more code you need to write (and trust me, when you build two separate modules for a game, you will write more code), the greater your mental model will have to be. While much of your code can be reused between the client and the server (especially if you use well-established programming techniques, such as object-oriented programming), at the end of the day, you need to manage a greater level of complexity.

• **Single point of failure and network congestion:** Up until now, we have mostly discussed the case where only a handful of players participate in the same game. However, the more common case is that a handful of groups of players play different games at the same time.

Using the same example of the two-player game of *Tic-tac-toe*, imagine that there are thousands of players facing each other in single games. In a peer-to-peer setup, once a couple of players have directly paired off, it is as though there are no other players enjoying that game. The only thing to keep these two players from continuing their game is their own connection with each other.

On the other hand, if the same thousands of players are connected to each other through a server sitting between the two, then two singled out players might notice severe delays between messages because the server is so busy handling all of the messages from and to all of the other people playing isolated games. Worse yet, these two players now need to worry about maintaining their own connection with each other through the server, but they also hope that the server's connection between them and their opponent will remain active.

All in all, many of the challenges involved in client-server networking are well studied and understood, and many of the problems you're likely to face during your multiplayer game development will already have been solved by someone else. Client-server is a very popular and powerful game networking model, and the required technology for it, which is available to us through HTML5 and JavaScript, is well developed and widely supported.

### Networking protocols – UDP and TCP

By discussing some of the ways in which your players can talk to each other across some form of network, we have yet only skimmed over how that communication is actually done. Let us then describe what protocols are and how they apply to networking and, more importantly, multiplayer game development.
The word protocol can be defined as a set of conventions or a detailed plan of a procedure [Citation [Def. 3,4], (n.d.). In Merriam Webster Online, Retrieved February 12, 2015, from http://www.merriam-webster.com/dictionary/protocol]. In computer networking, a protocol describes to the receiver of a message how the data is organized so that it can be decoded. For example, imagine that you have a multiplayer beat 'em up game, and you want to tell the game server that your player just issued a kick command and moved 3 units to the left. What exactly do you send to the server? Do you send a string with a value of "kick", followed by the number 3? Otherwise, do you send the number first, followed by a capitalized letter "K", indicating that the action taken was a kick? The point I’m trying to make is that, without a well-understood and agreed-upon protocol, it is impossible to successfully and predictably communicate with another computer.

The two networking protocols that we’ll discuss in the section, and that are also the two most widely used protocols in multiplayer networked games, are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). Both protocols provide communication services between clients in a network system. In simple terms, they are protocols that allow us to send and receive packets of data in such a way that the data can be identified and interpreted in a predictable way.

When data is sent through TCP, the application running in the source machine first establishes a connection with the destination machine. Once a connection has been established, data is transmitted in packets in such a way that the receiving application can then put the data back together in the appropriate order. TCP also provides built-in error checking mechanisms so that, if a packet is lost, the target application can notify the sender application, and any missing packets are sent again until the entire message is received.

In short, TCP is a connection-based protocol that guarantees the delivery of the full data in the correct order. Use cases where this behavior is desirable are all around us. When you download a game from a web server, for example, you want to make sure that the data comes in correctly. You want to be sure that your game assets will be properly and completely downloaded before your users start playing your game. While this guarantee of delivery may sound very reassuring, it can also be thought of as a slow process, which, as we’ll see briefly, may sometimes be more important than knowing that the data will arrive in full.

In contrast, UDP transmits packets of data (called datagrams) without the use of a pre-established connection. The main goal of the protocol is to be a very fast and frictionless way of sending data towards some target application. In essence, you can think of UDP as the brave employees who dress up as their company’s mascot and stand outside their store waving a large banner in the hope that at least some of the people driving by will see them and give them their business.
While at first, UDP may seem like a reckless protocol, the use cases that make UDP so desirable and effective includes the many situations when you care more about speed than missing packets a few times, getting duplicate packets, or getting them out of order. You may also want to choose UDP over TCP when you don't care about the reply from the receiver. With TCP, whether or not you need some form of confirmation or reply from the receiver of your message, it will still take the time to reply back to you, at least acknowledging that the message was received. Sometimes, you may not care whether or not the server received the data.

A more concrete example of a scenario where UDP is a far better choice over TCP is when you need a heartbeat from the client letting the server know if the player is still there. If you need to let your server know that the session is still active every so often, and you don't care if one of the heartbeats get lost every now and again, then it would be wise to use UDP. In short, for any data that is not mission-critical and you can afford to lose, UDP might be the best option.

In closing, keep in mind that, just as peer-to-peer and client-server models can be built side by side, and in the same way your game server can be a hybrid of authoritative and non-authoritative, there is absolutely no reason why your multiplayer games should only use TCP or UDP. Use whichever protocol a particular situation calls for.
Network sockets

There is one other protocol that we’ll cover very briefly, but only so that you can see the need for network sockets in game development. As a JavaScript programmer, you are doubtlessly familiar with Hypertext Transfer Protocol (HTTP). This is the protocol in the application layer that web browsers use to fetch your games from a Web server.

While HTTP is a great protocol to reliably retrieve documents from web servers, it was not designed to be used in real-time games; therefore, it is not ideal for this purpose. The way HTTP works is very simple: a client sends a request to a server, which then returns a response back to the client. The response includes a completion status code, indicating to the client that the request is either in process, needs to be forwarded to another address, or is finished successfully or erroneously (Hypertext Transfer Protocol (HTTP/1.1): Authentication, (June 1999). https://tools.ietf.org/html/rfc7235)

There are a handful of things to note about HTTP that will make it clear that a better protocol is needed for real-time communication between the client and server. Firstly, after each response is received by the requester, the connection is closed. Thus, before making each and every request, a new connection must be established with the server. Most of the time, an HTTP request will be sent through TCP, which, as we’ve seen, can be slow, relatively speaking.

Secondly, HTTP is by design a stateless protocol. This means that, every time you request a resource from a server, the server has no idea who you are and what is the context of the request. (It doesn't know whether this is your first request ever or if you're a frequent requester.) A common solution to this problem is to include a unique string with every HTTP request that the server keeps track of, and can thus provide information about each individual client on an ongoing basis. You may recognize this as a standard session. The major downside with this solution, at least with regard to real-time gaming, is that mapping a session cookie to the user’s session takes additional time.

Finally, the major factor that makes HTTP unsuitable for multiplayer game programming is that the communication is one way — only the client can connect to the server, and the server replies back through the same connection. In other words, the game client can tell the game server that a punch command has been entered by the user, but the game server cannot pass that information along to other clients. Think of it like a vending machine. As a client of the machine, we can request specific items that we wish to buy. We formalize this request by inserting money into the vending machine, and then we press the appropriate button.
Under no circumstance will a vending machine issue commands to a person standing nearby. That would be like waiting for a vending machine to dispense food, expecting people to deposit the money inside it afterwards.

The answer to this lack of functionality in HTTP is pretty straightforward. A network socket is an endpoint in a connection that allows for two-way communication between the client and the server. Think of it more like a telephone call, rather than a vending machine. During a telephone call, either party can say whatever they want at any given time. Most importantly, the connection between both parties remains open throughout the duration of the conversation, making the communication process highly efficient.

WebSocket is a protocol built on top of TCP, allowing web-based applications to have two-way communication with a server (The WebSocket Protocol, (December 2011). http://tools.ietf.org/html/rfc6455 RFC 6455). The way a WebSocket is created consists of several steps, including a protocol upgrade from HTTP to WebSocket. Thankfully, all of the heavy lifting is done behind the scenes by the browser and JavaScript, as we'll see in the next section. For now, the key takeaway here is that with a TCP socket (yes, there are other types of socket including UDP sockets), we can reliably communicate with a server, and the server can talk back to us as per the need.

Socket programming in JavaScript
Let's now bring the conversation about network connections, protocols, and sockets to a close by talking about the tools—JavaScript and WebSockets—that bring everything together, allowing us to program awesome multiplayer games in the language of the open Web.
The WebSocket protocol

Modern browsers and other JavaScript runtime environments have implemented the WebSocket protocol in JavaScript. Don’t make the mistake of thinking that just because we can create WebSocket objects in JavaScript, WebSockets are part of JavaScript. The standard that defines the WebSocket protocol is language-agnostic and can be implemented in any programming language. Thus, before you start to deploy your JavaScript games that make use of WebSockets, ensure that the environment that will run your game uses an implementation of the ECMA standard that also implements WebSockets. In other words, not all browsers will know what to do when you ask for a WebSocket connection.

For the most part, though, the latest versions, as of this writing, of the most popular browsers today (namely, Google Chrome, Safari, Mozilla Firefox, Opera, and Internet Explorer) implement the current latest revision of RFC 6455. Previous versions of WebSockets (such as protocol version 76, 7, or 10) are slowly being deprecated and have been removed by some of the previously mentioned browsers.

Probably the most confusing thing about the WebSocket protocol is the way each version of the protocol is named. The very first draft (which dates back to 2010), was named draft-hixie-thewebsocketprotocol-75. The next version was named draft-hixie-thewebsocketprotocol-76. Some people refer to these versions as 75 and 76, which can be quite confusing, especially since the fourth version of the protocol is named draft-ietf-hybi-thewebsocketprotocol-07, which is named in the draft as WebSocket Version 7. The current version of the protocol (RFC 6455) is 13.

Let us take a quick look at the programming interface (API) that we’ll use within our JavaScript code to interact with a WebSocket server. Keep in mind that we’ll need to write both the JavaScript clients that use WebSockets to consume data as well as the WebSocket server, which uses WebSockets but plays the role of the server. The difference between the two will become apparent as we go over some examples.

Creating a client-side WebSocket

The following code snippet creates a new object of type WebSocket that connects the client to some backend server. The constructor takes two parameters; the first is required and represents the URL where the WebSocket server is running and expecting connections. The second URL, which we won’t make use of in this book, is an optional list of sub-protocols that the server may implement.

```javascript
var socket = new WebSocket('ws://www.game-domain.com');
```
Although this one line of code may seem simple and harmless enough, here are a few things to keep in mind:

- We are no longer in HTTP territory. The address to your WebSocket server now starts with `ws://` instead of `http://`. Similarly, when we work with secure (encrypted) sockets, we would specify the server's URL as `wss://`, just like in `https://`.
- It may seem obvious to you, but a common pitfall that those getting started with WebSockets fall into is that, before you can establish a connection with the previous code, you need a WebSocket server running at that domain.
- WebSockets implement the same-origin security model. As you may have already seen with other HTML5 features, the same-origin policy states that you can only access a resource through JavaScript if both the client and the server are in the same domain.

For those who are not familiar with the same-domain (also known as the **same-origin**) policy, the three things that constitute a domain, in this context, are the protocol, host, and port of the resource being accessed. In the previous example, the protocol, host, and port number were, respectively `ws` (and not `wss`, `http`, or `ssh`), `www.game-domain.com` (any sub-domain, such as `game-domain.com` or `beta.game-domain.com` would violate the same-origin policy), and 80 (by default, WebSocket connects to port 80, and port 443 when it uses `wss`).

Since the server in the previous example binds to port 80, we don't need to explicitly specify the port number. However, had the server been configured to run on a different port, say 2667, then the URL string would need to include a colon followed by the port number that would need to be placed at the end of the host name, such as `ws://www.game-domain.com:2667`.

As with everything else in JavaScript, WebSocket instances attempt to connect to the backend server asynchronously. Thus, you should not attempt to issue commands on your newly created socket until you're sure that the server has connected; otherwise, JavaScript will throw an error that may crash your entire game. This can be done by registering a callback function on the socket's `onopen` event as follows:

```javascript
var socket = new WebSocket('ws://www.game-domain.com');
socket.onopen = function(event) {
    // socket ready to send and receive data
};
```
Once the socket is ready to send and receive data, you can send messages to the server by calling the socket object's `send` method, which takes a string as the message to be sent.

```javascript
// Assuming a connection was previously established
socket.send('Hello, WebSocket world!');
```

Most often, however, you will want to send more meaningful data to the server, such as objects, arrays, and other data structures that have more meaning on their own. In these cases, we can simply serialize our data as JSON strings.

```javascript
var player = {
    nickname: 'Juju',
    team: 'Blue'
};

socket.send(JSON.stringify(player));
```

Now, the server can receive that message and work with it as the same object structure that the client sent it, by running it through the parse method of the JSON object.

```javascript
var player = JSON.parse(event.data);
player.name === 'Juju'; // true
player.team === 'Blue'; // true
player.id === undefined; // true
```

If you look at the previous example closely, you will notice that we extract the message that is sent through the socket from the `data` attribute of some event object. Where did that event object come from, you ask? Good question! The way we receive messages from the socket is the same on both the client and server sides of the socket. We must simply register a callback function on the socket's `onmessage` event, and the callback will be invoked whenever a new message is received. The argument passed into the callback function will contain an attribute named `data`, which will contain the raw string object with the message that was sent.

```javascript
socket.onmessage = function(event) {
    event instanceof MessageEvent; // true
    var msg = JSON.parse(event.data);
};
```

---

**Downloading the example code**

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
Other events on the socket object on which you can register callbacks include `onerror`, which is triggered whenever an error related to the socket occurs, and `onclose`, which is triggered whenever the state of the socket changes to `CLOSED`; in other words, whenever the server closes the connection with the client for any reason or the connected client closes its connection.

As mentioned previously, the socket object will also have a property called `readyState`, which behaves in a similar manner to the equally-named attribute in AJAX objects (or more appropriately, `XMLHttpRequest` objects). This attribute represents the current state of the connection and can have one of four values at any point in time. This value is an unsigned integer between 0 and 3, inclusive of both the numbers. For clarity, there are four accompanying constants on the WebSocket class that map to the four numerical values of the instance's `readyState` attribute. The constants are as follows:

- **WebSocket.CONNECTING**: This has a value of 0 and means that the connection between the client and the server has not yet been established.
- **WebSocket.OPEN**: This has a value of 1 and means that the connection between the client and the server is open and ready for use. Whenever the object's `readyState` attribute changes from `CONNECTING` to `OPEN`, which will only happen once in the object's life cycle, the `onopen` callback will be invoked.
- **WebSocket.CLOSING**: This has a value of 2 and means that the connection is being closed.
- **WebSocket.CLOSED**: This has a value of 3 and means that the connection is now closed (or could not be opened to begin with).

Once the `readyState` has changed to a new value, it will never return to a previous state in the same instance of the socket object. Thus, if a socket object is `CLOSING` or has already become `CLOSED`, it will never `OPEN` again. In this case, you would need a new instance of WebSocket if you would like to continue to communicate with the server.

To summarize, let us bring together the simple WebSocket API features that we discussed previously and create a convenient function that simplifies data serialization, error checking, and error handling when communicating with the game server:

```javascript
function sendMsg(socket, data) {
    if (socket.readyState === WebSocket.OPEN) {
        socket.send(JSON.stringify(data));

        return true;
    }
}
```
return false;

};

Game clients

Earlier in the chapter, we talked about the architecture of a multiplayer game that was based on the client-server pattern. Since this is the approach we will take for the games that we'll be developing throughout the book, let us define some of the main roles that the game client will fulfill.

From a higher level, a game client will be the interface between the human player and the rest of the game universe (which includes the game server and other human players who are connected to it). Thus, the game client will be in charge of taking input from the player, communicating this to the server, receive any further instructions and information from the server, and then render the final output to the human player again. Depending on the type of game server used (we'll discuss this in the next section and in future chapters), the client can be more sophisticated than just an input application that renders static data received from the server. For example, the client could very well simulate what the game server will do and present the result of this simulation to the user while the server performs the real calculations and tells the results to the client. The biggest selling point of this technique is that the game would seem a lot more dynamic and real-time to the user since the client responds to input almost instantly.

Game servers

The game server is primarily responsible for connecting all the players to the same game world and keeping the communication going between them. However as you will soon realize, there may be cases where you will want the server to be more sophisticated than a routing application. For example, just because one of the players is telling the server to inform the other participants that the game is over, and the player sending the message is the winner, we may still want to confirm the information before deciding that the game is in fact over.

With this idea in mind, we can label the game server as being of one of the two kinds: authoritative or non-authoritative. In an authoritative game server, the game's logic is actually running in memory (although it normally doesn't render any graphical output like the game clients certainly will) all the time. As each client reports the information to the server by sending messages through its corresponding socket, the server updates the current game state and sends the updates back to all of the players, including the original sender. This way we can be more certain that any data coming from the server has been verified and is accurate.
In a non-authoritative server, the clients take on a much more involved part in the game logic enforcement, which gives the client a lot more trust. As suggested previously, what we can do is take the best of both worlds and create a mix of both the techniques. What we will do in this book is have a strictly authoritative server, but clients that are smart and can do some of the work on their own. Since the server has the ultimate say in the game, however, any messages received by clients from the server are considered as the ultimate truth and supersede any conclusions it came to on its own.

**Putting it all together – Tic-tac-toe**

Before we go crazy with our new knowledge about networking, WebSockets, and multiplayer game architecture, let us apply these principles in the simplest way possible by creating a very exciting networked game of *Tic-tac-toe*. We will use plain WebSockets to communicate with the server, which we’ll write in pure JavaScript. Since this JavaScript is going to be run in a server environment, we will use Node.js (refer to https://nodejs.org/), which you may or may not be familiar with at this point. Do not worry too much about the details specific to Node.js just yet. We have dedicated a whole chapter just to getting started with Node.js and its associated ecosystem. For now, try to focus on the networking aspects of this game.
Surely, you are familiar with *Tic-tac-toe*. Two players take turns marking a single square on a 9x9 grid, and whoever marks three spaces on the board with the same mark such that a straight line is formed either horizontally, vertically, or diagonally wins. If all nine squares are marked and the previously mentioned rule is not fulfilled, then the game ends in a draw.

**Node.js – the center of the universe**

As promised, we will discuss Node.js in great depth in the next chapter. For now, just know that Node.js is a fundamental part of our development strategy since the entire server will be written in Node, and all other supporting tools will take advantage of Node's environment. The setup that we'll use for this first demo game contains three main parts, namely, the **web server**, the **game server**, and the **client files** (where the game client resides).

There are six main files that we need to worry about for now. The rest of them are automatically generated by Node.js and related tooling. As for our six scripts, this is what each of them does.
The /Player.js class

This is a very simple class that is intended mostly to describe what is expected by both the game client and the server.

```javascript
/**
 * @param {number} id
 * @param {string} label
 * @param {string} name
 * @constructor
 */
var Player = function(id, label, name) {
  this.id = id;
  this.label = label;
  this.name = name;
};

module.exports = Player;
```

The last line will be explained in more detail when we talk about the basics of Node.js. For now, what you need to know is that it makes the Player class available to the server code as well as the client code that is sent to the browser.

In addition, we could very well just use an object literal throughout the game in order to represent what we're abstracting away as a player object. We could even use an array with those three values, where the order of each element would represent what the element is. While we're at it, we could even use a comma-separated string to represent all the three values.

As you can see, the slight verbosity incurred here by creating a whole new class just to store three simple values makes it easier to read the code, as we now know the contract that is established by the game when it asks for a Player. It will expect attributes named id, label, and name to be present there.

In this case, id can be considered a bit superfluous because its only purpose is to identify and distinguish between the players. The important thing is that the two players have a unique ID. The label attribute is what each player will print on the board, which just happens to be a unique value as well between both the players. Finally, the name attribute is used to print the name of each player in a human-readable way.
The /BoardServer.js class

This class abstracts a representation of the game of Tic-tac-toe, defining an interface where we can create and manage a game world with two players and a board.

```javascript
var EventEmitter = require('events').EventEmitter;
var util = require('util);

/**
* @constructor
*/
var Board = function() {
    this.cells = [];
    this.players = [];
    this.currentTurn = 0;
    this.ready = false;

    this.init();
};

Board.events = {
    PLAYER_CONNECTED: 'playerConnected',
    GAME_READY: 'gameReady',
    CELL_MARKED: 'cellMarked',
    CHANGE_TURN: 'changeTurn',
    WINNER: 'winner',
    DRAW: 'draw'
};

util.inherits(Board, EventEmitter);

As this code is intended to run in the server only, it takes full advantage of Node.js. The first part of the script imports two core Node.js modules that we'll leverage instead of reinventing the wheel. The first, EventEmitter, will allow us to broadcast events about our game as they take place. Second, we import a utility class that lets us easily leverage object-oriented programming. Finally, we define some static variables related to the Board class in order to simplify event registration and propagation.

```
The `Board` class exposes several methods that a driver application can call in order to input data into it, and it emits events when certain situations occur. As illustrated in the method mentioned previously, whenever a player successfully marks an available square on the board, the game broadcasts that event so that the driver program knows what has happened in the game; it can then contact each client through their corresponding sockets, and let them know what happened.

**The `/server.js` class**

Here, we have the driver program that uses the `Board` class that we described previously in order to enforce the game's rules. It also uses WebSockets to maintain connected clients and handle their individual interaction with the game.

```javascript
var WebSocketServer = require('ws').Server;
var Board = require('./BoardServer');
var Player = require('./Player');

var PORT = 2667;
var wss = new WebSocketServer({port: PORT});
var board = new Board();

var events = {
    incoming: {
        JOIN_GAME: 'csJoinGame',
        MARK: 'csMark',
        QUIT: 'csQuit'
    },
    outgoing: {
        JOIN_GAME: 'scJoinGame',
        MARK: 'scMark',
        SET_TURN: 'scSetTurn',
        OPPONENT_READY: 'scOpponentReady',
        GAME_OVER: 'scGameOver',
        ERROR: 'scError',
        QUIT: 'scQuit'
    }
};

/**
 * @param action
 * @param data
 * @returns {string}
 */
```
function makeMessage(action, data) {
    var resp = {
        action: action,
        data: data
    };

    return JSON.stringify(resp);
}

console.log('Listening on port %d', PORT);

The first part of this Node.js server script imports both our custom classes (Board and Player) as well as a handy third-party library called ws that helps us implement the WebSocket server. This library handles things such as the setup of the initial connection, the protocol upgrade, and so on, since these steps are not included in the JavaScript WebSocket object, which is only intended to be used as a client. After a couple of convenience objects, we have a working server that waits for connections on ws://localhost:2667.

wss.on('connection', function connection(ws) {
    board.on(Board.events.PLAYER_CONNECTED, function(player) {
        wss.clients.forEach(function(client) {
            board.players.forEach(function(player) {
                client.send(makeMessage(events.outgoing.JOIN_GAME,
                                          player));
            }));
        }));
    });

ws.on('message', function incoming(msg) {
    try {
        var msg = JSON.parse(msg);
    } catch (error) {
        ws.send(makeMessage(events.outgoing.ERROR, 'Invalid action'));
        return;
    }

    try {
        switch (msg.action) {
            case events.incoming.JOIN_GAME:
                var player = new Player(board.players.length + 1,
                                         board.players.length === 0 ? 'X' : 'O', msg.data);
                board.addPlayer(player);
                break;
            break;
        }
    }
The rest of the important stuff with this server happens in the middle. For brevity, we've only included one example of each situation, which includes an event handler registration for events emitted by the Board class as well as registration of a callback function for events received by the socket. (Did you recognize the `ws.on('message', function(msg){})` function call? This is Node's equivalent of the client-side JavaScript `socket.onmessage = function(event){}` that we discussed earlier.)

Of major importance here is the way we handle incoming messages from the game clients. Since the client can only send us a single string as the message, how are we to know what the message is? Since there are many types of messages that the client can send to the server, what we do here is create our own little protocol. That is, each message will be a serialized JSON object (also known as an object literal) with two attributes. The first will be keyed with the value of action and the second will have a key of data, which can have a different value depending on the specified action. From here, we can look at the value of `msg.action` and respond to it accordingly.

For example, whenever a client connects to the game server, it sends a message with the following value:

```javascript
{
    action: events.outgoing.JOIN_GAME,
    data: "<player nickname>
}
```

Once the server receives that object as the payload of the `onmessage` event, it can know what the message means and the expected value for the player's nickname.

## The `/public/js/Board.js` class
This class is very similar to `BoardServer.js`, with the main difference being that it also handles the DOM (meaning the HTML elements rendered and managed by the browser), since the game needs to be rendered to human players.

```javascript
/**
 * @constructor
 */
```
Chapter 1

/*
 var Board = function(scoreBoard) {
     this.cells = [];
     this.dom = document.createElement('table');
     this.dom.addEventListener('click', this.mark.bind(this));
     this.players = [];
     this.currentTurn = 0;
     this.ready = false;

     this.scoreBoard = scoreBoard;

     this.init();

};

Board.prototype.bindTo = function(container) {
    container.appendChild(this.dom);
};

Board.prototype.doWinner = function(pos) {
    this.disableAll();
    this.highlightCells(pos);
};

Again, for brevity, we have chosen not to display much of the game's logic. The important things to note here are that this version of the Board class is very much DOM-aware, and it behaves very passively to game decisions and the enforcement of the game's rules. Since we're using an authoritative server, this class does whatever the server tells it to, such as marking itself in a way that indicates that a certain participant has won the game.

The /public/js/app.js class
Similar to server.js, this script is the driver program for our game. It does two things: it takes input from the user with which it drives the server, and it uses input that it receives from the server in order to drive the board.

    var socket = new WebSocket('ws://localhost:2667');

    var scoreBoard = [
        document.querySelector('#p1Score'),
        document.querySelector('#p2Score')
    ];

    var hero = {};
    var board = new Board(scoreBoard);
board.onMark = function(cellId) {
    socket.send(makeMessage(events.outgoing.MARK, {playerId: hero.id, cellId: cellId}));
};

socket.onmessage = function(event){
    var msg = JSON.parse(event.data);

    switch (msg.action) {
    case events.incoming.GAME_OVER:
        if (msg.data.player) {
            board.doWinner(msg.data.pos);
        } else {
            board.doDraw();
        }
        socket.send(makeMessage(events.outgoing.QUIT, hero.id));
        break;

    case events.incoming.QUIT:
        socket.close();
        break;
    }
};

socket.onopen = function(event) {
    startBtn.removeAttribute('disabled');
    nameInput.removeAttribute('disabled');
    nameInput.removeAttribute('placeholder');
    nameInput.focus();
};

Again, it is noteworthy how DOM-centric the client server is. Observe also how obedient the client is to the messages received from the server. If the action specified by the server in the message that it sends to the clients is **GAME_OVER**, the client cleans things up, tells the player that the game is over either because someone won the game or the game ended in a draw, then it tells the server that it is ready to disconnect. Again, the client waits for the server to tell it what to do next. In this case, it waits for the server to clean up, then tells the client to disconnect itself.
Summary
In this chapter, we discussed the basics of networking and network programming paradigms. We saw how WebSockets makes it possible to develop real-time, multiplayer games in HTML5. Finally, we implemented a simple game client and game server using widely supported web technologies and built a fun game of Tic-tac-toe.

In the next chapter, we will take a look at the current state of the art in the JavaScript development world, including JavaScript in the server through Node.js. The chapter will teach you current techniques to manage the development cycle in JavaScript with workflow and resource management tools such as NPM, Bower, Grunt, and so on.
Where to buy this book

You can buy Multiplayer Game Development with HTML5 from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.