# How I Created “Pizza Peers”

Hi, I am Pandaqi, an indie game developer who almost exclusively creates local multiplayer games.

I’ve been trying to create those kinds of games for several years now, which means I’ve hit every obstacle and roadblock imaginable.

In this article series I want to explain …

* **Why** I created “Pizza Peers” (and the underlying system)
* **How** I created it (with some code, some nice images, and a bunch of text)
* And why I think it might be the best thing I ever made and will be creating many more games like these.

If you’re reading this as a programmer, you’re in luck: I made the full source code publicly available! (I even did my best to clean it up and comment everything nicely.)

**Pizza Peers on GitHub (TO DO: Turn into a link.)**

These are the different articles in the series:

1. The problems with multiplayer and how to solve (some of) them
2. Web Sockets + Node.js
3. Peer to Peer connections
4. Using Phaser.io (version 3)
5. Generating the city
6. Things to keep in mind (syncing players, disconnecting, smartphone weirdness)
7. Conclusion

## The problem with multiplayer

I want to create local multiplayer experiences for the whole family. The games must be accessible to all ages, to both gamers and non-gamers, and most of all: cooperative multiplayer.

(Not always, of course. I make single player games and competitive multiplayer. It’s just that cooperative is the best choice for families and non-gamers, because they don’t tend to go well with competitive environments …)

**First thing I ever tried**: one person using the keyboard, one using the mouse. As you might expect, this was a mess. (And didn’t scale to more than two players, obviously. Nobody is going to attach four mice (mouses?) to their computer.)

**Second try**: two people on the same keyboard. In fact, I even made games for three and four people on the same keyboard. While this worked and was kinda fun, it’s just very cramped and doesn’t *feel* fun. Pressing tiny buttons with letters on them is not an *intuitive* way to control a game.

**Third try:** controllers / joysticks / gamepads (however people want to call them). Controllers feel the most fun and intuitive when playing a game. The number of people with at least one controller is quite high (in my experience), and this scales perfectly to higher player counts.

I really like playing with a controller, and so does everyone around me (they are visibly having more fun if each of them gets their own controller), so this seems like the perfect option.

*However*, not everybody has a controller, and certainly not more than two. Additionally, to non-gamers a controller feels “intimidating”. They immediately get the feeling that this game is going to be difficult and that they won’t understand it.

(Additionally, the setup can be draining. Having to connect all the controllers to a system before getting to play a game, and having to *explain* the buttons, costs a lot of time and energy. By the time the setup is done, half the people have already lost motivation.)

**Fourth try:** what if … we could use people’s *smartphones* as the controller?

I know about the **Jackbox** games, but those are mostly (social) turn-based games.

I know about **AirConsole**, but that service uses a server that introduces noticeable delays, and the games on there are – in my probably worthless opinion – not that exciting or accessible. (Plus, it costs money. That’s not a criticism: it’s fair to ask money for such a service. It’s just that I don’t personally have the money to spend.)

Instead, I remembered an old technique that now seems long forgotten in the gaming world: **peer to peer**. (I’ll explain this more in the third article.)

If you are playing on the same Wi-Fi network, using a peer-to-peer connection is **as good as instant**. And indeed, in all the games I’ve played using my system, everyone immediately forgets that they’re playing over the internet because everything happens as if you were holding a controller.

Suddenly, everyone with a smartphone can join the game within 5-10 seconds! Most people *have* a smartphone and *understand* how it works, which means these games are not intimidating and do not require you to buy loads of controllers.

## So, what *is* “Pizza Peers”?

That’s what Pizza Peers is: **a cooperative multiplayer game in the web browser, where everyone can use their smartphone as the controller**.

More specifically, you’ll be running a pizza place and trying to prepare and deliver pizzas throughout the city.

To start the game, all you need to do is visit [**https://pizza-peers.herokuapp.com**](https://pizza-peers.herokuapp.com)on a computer. Press the “create game” button.

Now, everyone can whip out their smartphone and visit the same address in their browser. They simply enter the *room code* (displayed on screen) and a *fun username*, and then click “join game”.

It only takes 5-10 seconds to fully create and load the game. It also only takes 5-10 seconds for players to connect (which they can do simultaneously, by the way). No need to download or install anything, no delays, you’re playing in no-time.

I can’t stress enough how important this is. With my previous games, I’d need to ask, “do you want to try my game?”, and if they said yes, it would take a good 10-15 minutes to setup everything and get everyone ready.

Now I can ask that question, point everyone to the web address, and within a minute we’re already playing. Needless to say, I’ve already playtested this game more often than many of my other games.

## Why I wrote this article series

I would *love* to see a world with more games like this. Games you can play with anyone, games that are extremely easy to start and access, cooperative multiplayer with everyone being in the same room.

(Sure, you can also play remotely if the host screenshares his computer. But that will introduce delays and probably not make the game more fun.)

I’m sharing this project, my source code, and my ideas behind it to show people what is possible and to hopefully help other game developers.

**It’s not that hard.** Seriously, the code for connecting people and sending signals through peer-to-peer is only a few hundred lines. (Which includes error handling, comments, lots of whitespace because I like whitespace, etcetera.)

Additionally, I haven’t found anyone else talk about this or share these kinds of ideas. I might be the first one to do this, although I highly doubt that.

**It opens a whole new world of possibilities.**

A smartphone, for example, is not static (like a controller). You can *change* the interface during the game, you can send chat messages, you can shake the phone to do something in-game, you can even livestream everyone’s camera to the screen if you want. (Not sure how you’d use that in a game, but you *can*!)

I’ve tested this game on a nearly 10-year old iMac (as the host) and my own 5-year old smartphone (which was the cheapest available at the time). Surely everyone must be able to play these kinds of games :p

So, let’s get started! Topics like web sockets and Node.js servers, what an exciting time!

# [Pizza Peers] WebSockets + Node.js server

This is part 2 in my article series about how I created “Pizza Peers”. If you haven’t read the previous article, make sure to do so as it explains the basic idea behind everything.

So, first things first, we need:

* A server that hosts our game (and serves the game files)
* A server that receives connections from both the players and the computer, so that it can *connect* them directly. (Once connected, the whole game goes via peer-to-peer.)

Obviously, we’ll use the same server for both these things.

**IMPORTANT REMARK:** In these articles, I will not show the full code (as it’s too complicated and specific, thus not so good for teaching or explaining). Instead, I give a template and some pseudo-code where needed. If you want to implement these things yourself, check out the source code for all the details and exceptions.

## Node.js

For the server side, we’ll use Node.js. It is simple and small, it uses JavaScript (which we’ll also use in the game itself), and I have some experience with it.

Below is the template for this server. It simply sets up a server that servers both static files (which are the game files) and accepts websocket connections (which are needed to connect players with the computer)

<script src="https://gist.github.com/Pandaqi/1d5c9db5d60fb0612bd94cb6f29c6202.js"></script>

Of course, all the magic is going to happen in that “on(message)” block.

## Peer to Peer

Which messages do we need to send? For that, we need to understand how peer-to-peer works.

Instead of creating a connection with the *server*, a peer connection is a direct connection *between two devices*. So, once established, your smartphone has a direct link with the computer (that hosts the game), and vice versa. This is what makes it so incredibly quick and easy.

However, we cannot allow devices to just establish connections as they please. That would not be very secure.

Instead, there’s a handshake protocol we need to follow:

* Device A wants to connect with device B
* A creates a peer (on their side) and sends out an **offer**
* B receives the offer, creates its own peer, and formulates a **response**.
* Once A has received and validated the response, both are officially connected.

For generating the offer and response signals, I use the **simple-peer** library (TO DO: Link here). I don’t understand what it’s doing with those signals or what all the information means, and neither do you. I just pass the signals along.

Speaking about that: we’ve encountered an issue. A needs to send an offer to B. But … they are not connected yet. How is A going to find B?

That’s where our WebSockets come in!

## Signaling server

In order to exchange signals, we turn our server into a so-called “signaling server”.

The idea is very simple:

* A generates a signal and sends it to the server.
* The *server* determines the intended recipient and relays the signal.
* B receives it, creates a response, sends it back.
* The *server* determines that the response must return to A.
* And voila, peer to peer connection!

So, at our server, we need a way to receive and pass on messages.

In my case, this is even more simplified:

* The *player* (with the smartphone) is always the initiator of the connection.
* The *computer* (that hosts the game) is always the one responding.

See the code below for the basic structure of this system. (It will make even more sense once we’ve created the client side.)

<script src="https://gist.github.com/Pandaqi/bf2c6215922a302a042126bb60c245c3.js"></script>

In fact, this is almost *all* the code on the server. The only thing I added in the real game is more robust error handling and handling some extra cases/exceptions.

## Client side - Websockets

So far, we’ve created the server only. It accepts web sockets and passes along signals … but we still need a client side to actually create those web sockets and signals.

I assume you know how to set up a basic HTML page. If not, check out index.html in my source code.

Then, make sure the JavaScript below is run (*once the page has finished loading*):

<script src="https://gist.github.com/Pandaqi/33474e8bda094296af95e6b8d53a367d.js"></script>

This bit of code opens the web socket connection, then listens for responses from the server. When necessary, it creates a new peer.

The “peer.signal(…)” bit will be explained soon. Essentially, we just pass the “signal message” directly into the peer and let it formulate its response. (Remember when I told you that I don’t know what’s inside those signals? Those things are put in here.)

**NOTE:** All messages are JSON. However, we cannot (and don’t want to) send objects over the internet. So, before sending, we must always *stringify* the object. At the receiving end, we always *parse* it, so it returns to the original JSON object.

## Almost done …

All the code above still does not complete our system for connecting – so don’t try to run out – but we’re very close.

This is what we’ve achieved so far:

* A server that servers our game files.
* A server that accepts socket connections, gets messages, and then relays them to the right connection.
* A client side that connects with the server, and also sends/receives the right messages, and creates the proper peers when needed.

All that’s left to do, is actually create the peers. For that, see you at part 3!

# [Pizza Peers] Peer to peer connections

This is part 3 in my article series about how I created “Pizza Peers”. If you haven’t read the previous articles, be sure to do so!

As stated earlier, I used the **simple-peer** (TO DO: link here) library. It’s free, it’s small, just grab it from GitHub and include it at the top of your index.html page.

In the code above, you saw the function “createPeer()”. In this article we’ll actually write that function.

## Creating peers

The code below creates the peer and then attaches the proper listeners (just like everything we’ve done before).

The “on(data)” call is where all the magic happens. As said before, once the connection is established, *all* communication goes via that listener and *none* of it uses the server anymore.

<script src="https://gist.github.com/Pandaqi/9553ee6e7646bc140bd4c0f77bcba4f4.js"></script>

At the top of the function you can see the actual peer being created.

The option “initiator” is true for players (because they *initiate* the connection), and false for the computer.

**NOTE:** The player only creates a single peer and then tries to connect with the computer. The computer creates a *new* peer for every player that wants to connect! That’s also why we can save variables on the peers relating to the specific player (such as its username or index in the game), because every player has a unique peer.

Remember: peer-to-peer is always a direct two-way connection between two peers. If you add a third player, for example, and want to connect everyone with everyone, you’d need to create *two peers per player*.

(I emphasize this, because it’s the reason it took me a whole day to get simple peers working. I didn’t fully understand what it did, what the “initiator” meant, and what ICE servers are. So, read on before you make the same mistake!)

## ICE Servers

The option “iceServers” requires some more explanation.

As said before, allowing any device to reach any other device is not very secure (and sometimes very hard to do). Thus, in some cases, you need so-called **STUN** and **TURN** servers to establish the connection.

* A STUN server basically acts as a mirror and allows a computer to find out its own IP-address, by bouncing a signal off of it.
* A TURN server is simply a middleman: it relays your signals directly to the other person.

As such, STUN servers are free and easy to get, because they barely do anything. TURN servers, on the other hand, will relay tons of data every second and are very expensive.

Luckily, these servers are rarely needed (perhaps 85-90% of the connections work without them).

Even better: **if you are on the same Wi-Fi, you should never need them**. You can just leave this option empty and it should work. (In the final version of Pizza Peers, I did acquire a free STUN and TURN server, just to cover all bases. But it’s not necessary for a *local* game.)

## What now?

All the code seen so far should be enough to connect to a server (and serve the game files), create a room, and allow players to create peer-to-peer connections within that room.

This is just the basic structure for networking. There’s no game or anything yet.

That’s our next stop along this adventure!

We’ll now look at the following elements:

* Starting and managing a game using Phaser.io
* Sending data from the game through a peer.
* The reverse: receiving data in a peer and sending it to the game.
* Creating the mobile interface => how to listen for input and other things to keep in mind.
* Creating the actual game => I won’t talk about everything, just the interesting bits like random city and kitchen generation

See you at part 4!

# [Pizza Peers] Using Phaser.io

This is part 4 in my article series about how I created “Pizza Peers”. If you haven’t read the previous articles, be sure to do so!