ECE 493 Final Report

Arun Woosaree

April 16, 2021

1 Problem domain

The goal of our project is to provide a fun multiplayer social game that provides an experience similar to Pong, but is easily accessible with modern technology. Imagine you have a bunch of friends that want to play a game together, but you all are physically separated. Furthermore, your friends do not want to install any new software on their computers. You and your friends each have a computer with either Firefox or Google Chrome installed on it and internet access. If you ever find yourself in this situation, this game is for you! Not only does our project provide a Pong-like experience, it also adds extra features which adds a unique spin on the game for extra fun. More than two players can participate, and also there are eleven fun power ups to choose from. Users can earn experience points, unlock skins, and compete for the most experience points with the leaderboard feature. Unlike most other games, there is no software to download or install. Most people already have a modern web browser like Firefox or Chrome. All a user needs to do is to visit polypong.ca and start playing. The experience is as frictionless as possible. Users are not even required to sign up before playing a game. Users can enjoy the game as a guest, and sign up later if they choose to.

PolyPong does not solve any huge problems in its own right, however, it does fit within the realm of gaming. It could be argued that one of the things that PolyPong solves is (at least temporarily) boredom. Humans have enjoyed playing games for a really long time. I'm not an anthropologist, but I think it is fair to say that modern humans have been playing games for a really long time, and that playing games is how many of us have kept ourselves busy and entertained. Games are a way to bring people together,

and we saw this happen just recently when the COVID pandemic hit: Steam (an online platform which distributes games) beat its highest concurrent players record multiple times last year. In a time when we were forced to isolate ourselves, we found ourselves more connected than ever by playing games together. PolyPong is another one of those things that friends can use to spend time together.

One problem we had to deal with was sharing state across multiple computers. A synchronization algorithm had to be developed. We did not invent anything new or groundbreaking here. We instead used a tried and true method for multiplayer game synchronization – the Dead Reckoning Algorithm. This allowed for clients of the game to predict the future state of the game and make small corrections in an effort to hide the latency of the connections between each client and the server.

2 Existing Solutions

A quick google search for "multiplayer pong" reveals that there are a plethora of online projects which already exist that allows one to play a multiplayer game similar to pong. Here are a few examples:

- `https://pong-2.com/` This has multiplayer online play, a single player mode, offline play, online multiplayer, and the ability to change the colours of the baddle, background and ball. However, this does not have power ups like our game. Furthermore, the online multiplayer feature is restricted to two players, while we have no such restriction
- `https://play.google.com/store/apps/details?id=com.AvidGames.Pong&hl=en_CA&gl=US` This requires an Android phone, and an app download. Also, it does not seem to provide online multiplayer with more than two players at once like we do, nor does it appear to have power ups.
- `https://github.com/pstefa1707/multiplayer-pong` This also seems to have a single player mode and online multiplayer. Like the other examples, there is no option to play a game with more than two players simultaneously.
- `https://github.com/HarshdeepGupta/multi_pong` This appears to work with up to four players, while our game allows for more than four

players to play in a single game. Also, it requires Java to be installed, while our game just runs in the browser.

- `https://github.com/Eisah-Jones/Multiplayer-Pong/tree/master/Multiplayer-Pong` This one seems to have a chatroom feature, which we do not have. However, this game seems to have a limit of four for the maximum amount of players that can play at once. Also, this game needs to be compiled, or a binary needs to be downloaded before it can be enjoyed, and the setup is not as simple as just going to a website like our game.

  A common trend among these examples seems to be that most of them have the limitation that the games usually have a set maximum number of players. In our game, we have no such limit, and 6, or even 12 players can easily play a game together. (The game does get more and more chaotic as more players join, and at a certain point, it does get impractical to play a game). However, we did not implement any single player mode in our game, while other alternatives usually have a single player mode with a computer opponent controlling the other paddle. I also found some games that seem to work offline, while our game requires the player to always be connected to the internet.

3 Our Solution

We wanted our game to be easily accessible, so that users can pick up the game, and start playing it in just a few seconds. This helped us narrow down our choices for what programming language to write our application in: The best way to make our application work on almost any device with next to no setup is to make it so that it can just run in a browser. Most modern devices have a web browser based on Google Chrome of Firefox pre-installed. So, if we make a web application, the user does not have to download any new program, or install anything. They can just visit a website and start playing right away!

The most popular language for web applications is JavaScript, however, we chose to go with TypeScript which has recently gained a lot of popularity. Created by Microsoft in 2012, TypeScript is a superset of JavaScript, which means that you can still write normal JavaScript code for your web application, however, you can also opt to use additional features, which helps to

avoid common pitfalls in JavaScript. One of the main benefits of using TypeScript over JavaScript is its type checking system, which allows for types to be statically checked before the code is transpiled to JavaScript. This helps to avoid errors such as referencing a null object by accident, or attempting to access a field that does not exist, because the programmer may not know the exact structure of the data. With TypeScript, one can write 'interfaces' for an object, which defines what fields should be available, and if the programmer does something that may cause an error down the line, the static analyzer can warn the programmer. We decided that we like these extra features that TypeScript has over JavaScript, so we went for it.

Alternatively, we could have chosen to write our code in a language that compiles to WebAssembly, like Rust. WebAssembly is a binary instruction format for a stack based virtual machine that runs in the browser. Rust is a modern low-level systems programming language with manual memory management, similar to C++. One of its main draws is its notorious compiler, which can guarantee memory safety using its infamous borrow checker which validates references to objects in memory and spits out errors, frustrating programmers everywhere when they write unsafe code. It manages to guarantee memory safety, while being on par with C++ in terms of performance. Similar programs written in C++ or Rust usually trade blows in terms of performance. Rust can be compiled to WebAssembly, so it would be totally feasible to write our game in that language. This would also bring some of the same benefits that a typed language like TypeScript has like static type checking, while offering other benefits like better performance. This language would be an excellent choice if we needed to squeeze our every bit of performance possible for our application. Luckily, drawing paddles, balls, and handling collisions is not that demanding for modern computers these days and we can afford a performance hit and instead use a language with garbage collection like JavaScript/TypeScript to make our jobs a bit easier. The borrow checker is wonderful, however, given the limited time for working on this project, we decided against it because it is easier to get something running and working when you don't have to worry about manual memory management.

Clearly, at this point, TypeScript seems to be in the winning spot for our language of choice, at least for the code running in the browser. Next, we needed to make a decision about how the game state would be synchronized between different players. We could either go with a peer-to-peer design where all of the code is client-side, or we could go with a client-server model.

Most online multiplayer games I have played personally have used this client-server model (Among Us, Team Fortress 2, Rocket League, <Insert Latest Battle Royale Game Here>...). Also, I have found from personal experience that peer-to-peer multiplayer games tend to be a bit more finnicky. For example, games like Super Mario Smash Bros. Ultimate and Mass Effect 3 Multiplayer tend to have issues when a peer loses their connection, and some players can get an unfair advantage over others if they are acting as the host, since they get the most up-to-date information.

We chose to go with a client server model over a peer-to-peer model, partially because of these past experiences as a consumer of multiplayer video games. We also thought it would be easier to go with a client-server model over deciding some consensus algorithm for the peer-to-peer method. The server is the authoritative source. This also allows us to keep track of user statistics in a centralized location, which we need to satisfy our functional requirements in the Software Requirements Specification. In order to provide a leaderboard feature, we need to store the ranking of each user in a central location, so the decision to go with a client-server model for the game was obvious.

Now that we decided to go with a client-server model, we needed to decide what language to write the backend in. From previous experience, I have found that one of the most frustrating things as either a frontend or backend developer is when the frontend developer and the backend developer have different ideas of what the data looks like, or the data models are out-of-sync. The frontend might be expecting for something that the backend does not provide, while the backend developer might not know what the frontend developer wants. I have found in previous projects that when the backend and frontend are written in the same language, I usually don't have to worry about this happening, because the frontend and backend can depend on the same data models. When the types or interfaces for a piece of data is updated for the backend for example, the frontend also gets that update immediately since it also imports the same dependency. When something breaks because of the data model, the frontend and backend developers notice immediately, and the problem tends to get solved quicker, instead of one party saying something to the effect of "it's a backend/frontend problem". The obvious choice for the backend code then, is also TypeScript.

At this point, it was clear that TypeScript is our language of choice – for both the backend and the frontend code. It was time to choose what frameworks to use. For the frontend, we decided to go with Svelte. The

obvious choice would be to go with React. It's the most popular JavaScript frontend framework today, and for good reason. I also have previous experience working with React. However, that would be boring. I convinced my team to use Svelte (mostly because I wanted to try something new). It's been under my radar for a while, and it looks cool. Instead of having a virtual DOM like React, the code is compiled to vanilla JavaScript, HTML, and CSS. Getting rid of the virtual DOM helps with performance: `https://svelte.dev/blog/virtual-dom-is-pure-overhead` However, while this is cool from a technical standpoint, it does not really mean much for us. For this project, the difference would not be noticeable. I also thought that Svelte would be easier for the rest of my team to learn, since its syntax is similar to vanilla JavaScript, HTML, and CSS, while React has some quirks that might contribute to a steeper learning curve like JSX syntax, and its lifecycle methods. JSX is HTML-like syntax in JavaScript code, which can be confusing since it is not the same as HTML. From the examples I have seen, Svelte code was simpler. Also, it was trivial to copy the initial mockups that Micheal had made in vanilla HTML and CSS, paste it in Svelte, make some minor tweaks, and build on top of it. I know for a fact that if we had chosen React, that process would have been a bit more involved.

The next choice was to decide what technologies the backend would use. The obvious choice here would be Node.js with the express framework. It's an industry standard, and works extremely well. However, I wanted to experiment and play with a newer runtime: Deno. Deno is written by the same guy who made Node.js: Ryan Dahl. He announced it in 2018 during a talk where he explained some of the things he regretted about Node.js. Deno has some security features that come out of the box, unlike Node.js. For example, it does not have network access, environment variable access, or access to the filesysttm by default. These features are opt-in only with a command-line flag. In contrast with Node.js, these permissions are implicitly allowed. Overall, we had no good reason to use Deno over a more established and popular runtime like Node.js, but it did not matter too much for what we were doing. The functionality we needed was there, and there was a routing library like express for Deno called oak. Plus, Deno has a really cute dinosaur logo:

With the frontend and backend language and frameworks chosen, we now had to decide how the client and server would communicate. When a game is active, we need a connection with low overhead and low latency. It would also ideally be bi-directional and the server or client would be able to send a message whenever it wants to, while also being ready to receive a message at any time. Because we chose to make our project run in the browser, our choices basically boiled down to using either WebSockets or WebRTC. WebRTC is usually used for peer-to-peer applications like video calling, however, it also supports the client-server model. It uses the UDP protocol which has the benefit of really low latency, however, because it uses UDP, the connection is not reliable. Messages can be lost, and we would need to account for that in our code. WebSockets on the other hand, use the TCP protocol, and are much easier to use and understand. TCP makes sure that each message is

sent and received using acknowledgements. It also ensures that the messages are received in the same order that they are sent, which helps keeps things simple. The WebSocket protocol is also reasonably fast, and we found it much easier to use than WebRTC.

With the major architectural decisions made, it was time to decide what features we would add to our game. What would differentiate us from similar games? We got a hint of this in the section above, talking about the existing solutions. First, we wanted to give the player a sense of progression, should they choose to log in to our system. They can earn experience points, and unlock fancy new skins. Skins do not affect the functionality of the game, but it is well-known that people love to customize things in games with skins in games like Counter Strike: Global Offensive, Fortnite, and Rocket League. For example, at the time of writing, this shiny knife skin for Counter Strike can be bought for $420.69 USD:

We decided against implementing any sort of monetization for our game. For one, dealing with payment information sounds like a nightmare. Also, we thought it would feel more rewarding if users actually earned the skins that they get to use by playing the game and earning experience points.

We also added a leaderboard feature so that users can compete amongst themselves, reaching for the highest score by playing the most games. Users should also be able to see statistics like their win/loss ratio and total games played. We did not find an alternative game that already has this feature. Powerups were another thing we wanted to add to differentiate ourselves from the classic Pong experience. Surely, there are other games out there with paddles, balls, and power ups, but in my quick searching, of the ones that were online and working, none had power ups. This was a similar story for the ability to play with more than four other players online.

For the login functionality, we initially were going to do an e-mail based magic link signin, like Slack does. We instead opted to go for an OAuth authentication flow because it is more secure than sending an authentication token over email. We chose not to store user passwords so that we do not need to have any sensitive information in our database. Instead, we use a JSON web token (JWT) issued by a trusted party (the OAuth provider), and use a shared secret to verify the token and the data in its payload. We decided to use Auth0 as our OAuth provider because it was the first OAuth provider we found on Google, and it seems to have a generous free tier.

For our database, it really did not matter what we picked. We just needed a simple key-value store to record things like the player's username, their experience points, and their currently selected skin. We went with MongoDB since it is a popular key-value database. We had to make sure that a user could not set their skin to a colour that was not unlocked yet, so we check that a user has sufficient experience points before doing the transaction. We also ensure that no user can set someone else's skin by modifying their own request. This is done by verifying the JSON web token issued to the user, which is also passed to the server. That way, the only skin that a player can change is their own.

For hosting the frontend, we went with the easiest to use service that allows one to host static files. Svelte allows the project to be minified into a bundle of optimized code, which is then uploaded to a server where the main `index.html` file is served. We happened to use Cloudflare Pages, but we could have easily also used something like GitHub Pages, Netlify, or Vercel which all offer the same functionality. Because we are just serving the files statically, and we chose to use client-side routing (as opposed to server-side), we opted to use hash based routing. This means that instead of going to a path like `/lobby` on the website, you would instead go to `/#/lobby`. This has the advantage of not requiring server-side logic for routing.

For hosting the backend, we chose to use Cybera. Cybera is a non-profit organization in Alberta which provides free computing resources to students and entrepreneurs. Alternatives include services like Google Cloud Platform, Linode, DigitalOcean, Vultr, etc. We went with Cybera, because it is free for us to use and experiment with as students. We chose to package the backend in a Docker container to make deployment to Cybera easy. Deno is not in the Ubuntu software repositories by default, so instead of getting our backend to work with a specific setup, we made a Dockerfile so that the deployment of the backend is easily reproducible.

4 Potential Impact on Society and the Environment

To be honest, I don't see our project significantly changing someone else's life for better or worse. Unless this game goes viral, the societal and environmental impact will both be relatively low. It's a simple, fun game that friends could get together and play for a bit. The application is not particularly demanding and we found that the CPU usage for machines running this application does not increase significantly while the game is running. Therefore, the amount of energy used by players is pretty low.

Hosting our project online as a website accessible at polypong.ca undoubtedly has an environmental impact. Servers must be kept online, so that the application is available for anyone to access. Our frontend is hosted using Cloudflare Pages, a service offered for free by Cloudflare. Our domain is also registered with Cloudflare, and we are using their DNS services. Because this is an on-demand service, this means that the servers used to run our frontend code does not always need to be active, if users are not using the website. These resources can be used by other users of the Cloudflare Pages service when our demand is low. Furthermore, Cloudflare appears to be a company which is conscious about their impact on the environment. For example, in 2019, they purchased Renewable Energy Certificates to match their electricity use for all of their data centres and offices around the world: `https://blog.cloudflare.com/the-climate-and-cloudflare/`

Our backend code is hosted on Cybera, a local nonprofit organization in Alberta. Unfortunately, due to how we designed the project, the backend server must run constantly to be ready for a client to connect to it. Also, this

code does not automatically scale back when the demand is low. Fortunately, it is not running on dedicated resources, as the CPU and memory resources are shared, so other users of the service can use the resources when we are not.

In my searching, I did not find any information about efforts Cybera is making to lessen their impact on the environment, however, we did find that Cybera is using data science to support green tech solutions: `https://www.cybera.ca/cybera-uses-data-science-to-support-green-tech-solutions/`

5 My Role

For this project, I found myself acting as a senior developer of sorts. I got to make decisions about which technologies we were using, and I found my teammates asking for advice on best practices and such, because I have previous experience with Javascript frameworks like React and Node.js. My team trusted the decisions I was making. Most of the time, when we wanted to start working on a new feature, I would be the one to make some boilerplate code that we would then be able to build on top of. This also had the benefit of showing others examples of how to write code for the frameworks we were using.

I worked mainly on the backend and database side of things. I also touched a fair bit of the frontend code, mainly hooking up core functionality and making sure that the frontend can communicate properly with the server. Basically, I touched almost every element of the project that was non visual. Things like fetching data from the server and displaying them, minus the looking pretty part. I also worked entirely on the authentication system. This involved managing the dashboard on Auth0, and writing code to do the login flow on the frontend. On the backend side of things, this involved code like requiring certain client requests to be authenticated and verifying the JSON web token provided in the request. I also worked on getting our project deployed so that we can play the game on polypong.ca. This involved domain registration, administering DNS records, deploying the frontend to Cloudflare Pages, writing Dockerfiles, and deploying the backend to Cybera. For the database side of things, I wrote the queries and helper functions which would be used in other parts of the application. I also worked on designing most of the tests like the database and server tests, while Micheal did the power up tests.

Micheal mainly focused on the frontend, UI design, and making features which I added look pretty. He also had to work a bit in the backend when he wanted to add features to the frontend. As a result, Micheal has a really good understanding of how the frontend works, and a solid understanding of how the backend works. Eventually, we also had to refactor the code and we pair programmed together, fixing bugs, adding features, and fixing more bugs. He came up with the initial UI prototype for the frontend, a large portion of which we have tweaked and kept in the final product. He also worked on getting the game to render on the HTML canvas, and did some geometry math to figure out how to render an n-sided polygon on the canvas, rotate the canvas, draw shapes, etc.

Micheal also figured out how to do collision detection, input handling, and designing the game loop. I pair programmed with Micheal a lot over the course of the project, getting things between the frontend and backend in sync, and making sure they communicate with each other. Together, we got a basic game working, one without synchronization. "Basic game" meaning that we got multiple players to connect to the server, get their paddles to move on each others' screens, and a ball moving. However, at that point, there was no synchronization, and although we could see the ball moving in the same direction on all screens, the movement was jittery because of the lack of synchronization. Micheal also implemented a majority of the eleven powerups in the game. I helped to implement some when we were pair programming as well.

Joshua mainly worked on the synchronization algorithm. The algorithm implemented was the dead reckoning algorithm. Basically, each client predicts what will happen in the game, and small corrections are made, which helps make the game appear smoother. There was also one day where we pair programmed together for a bit, and another few days leading up to the project demo date where we programmed together, along with Micheal to get the remaining power ups implemented. Joshua also wrote the initial code we used for the 'Add Ball' powerup. His main contribution was implementing the Dead Reckoning Algorithm, which makes the clients predict the movement of the paddles so that the game appears to be more smooth and less jittery. This involved simulating the paddle movement for opponents, and also predicting the movement of the ball. Overall, Joshua's involvement in the project was limited.