Project Report

Full-Stack Networked Physics Synchronization

SEG3904

Innovation Research Project

Professor: Daniel Amyot (damyot@uottawa.ca)

April 20, 2019

Dominic Roy-Stang

Student Number: 7483706

TABLE OF CONTENTS

INTRODUCTION

Push Pong was started with the broad objective of learning the process involved in the development of a full-stack web application that uses websockets to communicate and synchronize a shared state between several concurrent users. Developing a simple multiplayer game where players interact with a shared resource seemed like a good proof of concept that would require the aforementioned skillset. Therefore, it was decided that creating a modified online multiplayer version of the classic game *pong* would be a perfect candidate for such an application. The game's concept is simple enough that the development time would not be entirely spent on building the game itself, but still complex enough that synchronization issues would be very noticeable by players.

PRIOR RESEARCH

Before starting the development phase of my project, I did some research to validate my choice of technologies. In particular, I focused on the area that I knew the least about: networked physics and netcode. A big part of this project was understanding how multiplayer games achieve a high level of synchronicity between the physics engines of their players even when the latency is quite high. To ensure that this part of the work would not be done for me, I avoided using a *game engine*. However, I did read up on what game engines and multiplayer game development frameworks do to manage synchronized physics state. Among many other sources, this documentation from lance.gg, explained that multiplayer games are mostly synchronized via snapshots, interpolation, and extrapolation. Networked physics synchronization is further discussed later in this report.

SCAFFOLDING

As I established the main requirements for the project, I started developing a *hello world* for each of the technologies that would be used for the project without necessarily integrating everything together. Once I was fairly confident that the technologies would work as intended, I started integrating everything together.

This phase of the project took less time than anticipated. I managed to quickly put together a basic node.js backend Express server that was capable of communicating with a React frontend via websockets on localhost using Socket.io. I already had a very basic knowledge of React and Express, so this part went without any major issue. The documentation for Socket.io made it easy to get a quick proof of concept (hello world) setup between the two.

CONTAINERIZATION

Once I had my basic backend and frontend services set up, I decided that it would be a good idea to containerize them so that I could reliably run them in any environment. This seemed like a perfect use case for Docker.

I had never created a docker container before, so creating containers for the services required a fairly significant amount of reading. Fortunately, I found a very detailed 13-part Tutorial Series on Medium that allowed me to gain the knowledge I needed to create Docker images for each of my services.

Once I got my images set up, I created a *docker-compose file* to allow any environment that has docker and docker-compose installed to instantiate the containers in a single command. This created a great developer experience because it reduces the time a developer needs to spend installing the correct dependencies, and reduces the time needed to run the code. Since I was the only one working on this project, I did not benefit much from the containerization of my services, but this skill will certainly be useful later in my career when trying to simplify the onboarding and workspace setup process for new developers.

CLOUD HOSTING

One thing I really wanted to get a good grasp of with this project was the usage of cloud hosting solutions. Cloud providers such as Google Cloud Platform (GCP) are used by many companies in the software development industry to quickly get scalable infrastructure for their products. For this reason, I wanted to try my hand at using professional-grade solutions even if my product would likely never end up being used by more than a few people at once. From what I have heard from several professional software developers, cloud services are an integral part of internet-facing product development, so I used this project to get an introduction to these tools.

App Engine

The first step was to bring my frontend and backend services to the cloud. Google App Engine was used to host the services. While containerization didn't provide a significant benefit for local development, it did end up paying off when it came to hosting the frontend and backend services on Google Cloud Platform's App engine service. As it turns out, GCP has several tools that make use of docker containers. Thanks to my prior work to containerize my services, I was able to rapidly convert them to App Engine services by simply adding a very basic app.yaml file to each of the services, pushing my images to Google's Container Registry, and deploying the images.

Custom Domain Name

At this point, I was able to access my basic website via push-pong.appspot.com. However, I also wanted to understand how to get my own domain name just like any normal company would do. To accomplish this, I purchased pushpong.xyz from Namecheap for the very low price of $1 per year. I then performed modifications to the DNS records to point to my google hosted frontend service by following a GCP guide on mapping custom domains. It took a several attempts to get pushpong.xyz to point to the frontend service, and backend.pushpong.xyz to point to the backend service due to the lacking documentation regarding working with subdomains. This part was especially painful because DNS changes can take up to 24 hours to take effect, so I was limited to one new attempt per day. Fortunately, I eventually got it working as desired, and even managed to get SSL certificates set up so that https would be working and enabled by default on both services.

Continuous Integration

By this point, I had the user-facing website working perfectly, but updating the website was still a manual process. This setup would probably disappoint a DevOps engineer because of the number of manual steps involved in releasing a new public-facing version. I needed some continuous integration. Hence, I learned how to use Google Cloud Build. Cloud build hooks into a repository and triggers operations when changes occur. I set it up so that it triggers the following steps on every push to the master branch on my GitHub repository:

- Build backend image
- Build frontend image
- Push backend image to the registry
- Push frontend image to the registry
- Deploy backend service on AppEngine
- Deploy frontend service on AppEngine

Some people go far beyond those few steps, but I figured that this short list of steps should be sufficient for the scale of my project.

Development & Production Environments

Now that the build process was working, there was only one final problem. Testing communication between the two services worked perfectly on localhost, but failed when running on GCP. The issue was due to the fact that the backend IP was hardcoded to localhost on the frontend. Obviously, this would not work in production code running on GCP because the two servers have different, publicly exposed IP addresses. One way to solve this issue would simply be to hardcode the IP address of the backend, but this would result in terrible developer experience due to the fact that one would have to deploy to master to verify that any code involving communication between the two services works properly. Evidently, a better solution would be to somehow let the code know if it is in development mode or production mode. However, this was much easier said than done.

The basic idea for my solution to this problem was the following. If the code is run directly or by using docker-compose, it would be assumed that the code is running in development mode and the backend host and port would be set to *localhost:5000*. If the code is run from the docker files directly (not from docker-compose), the backend host and port would be set to *backend.pushpong.xyz*.

This was implemented by using two page servers: the default react page server when running in development mode, and a custom page server for production mode. The only tangible difference between the page server and the regular server is that it would inject the backend host, port, and protocol as variables into the HTML of the page before sending the page to the browser.

This worked when testing locally. I was able to see that the frontend had different addresses for the backend server depending on how I was running the code. However, I ran into some issues when the code was running on Google Cloud Platform. For some reason, the website returned an error code 503 when trying to access the frontend site, probably due to some Google App Engine requirement for hosting servers.

In the end, my professor suggested that I move my focus to the networking and synchronization aspect of the project, so I put this part of the project aside. I do believe that my approach to solving the problem was valid. It is very likely that I was one GCP configuration away from being able to fix this issue on the cloud side. However, with the quality of the documentation for GCP, fixing this one issue could have potentially taken over 12 hours, so I agreed that shifting my focus was the best choice given my limited development time.

SELECTING A PHYSICS ENGINE

I had never done any real game development before starting this project, so it was hard for me to know what to look for. All I knew was that I wanted the game to be playable on a browser, so I looked up the most popular JavaScript physics engine. After some research, Matter.js stood out as the most recommended physics engine. Hence, I implemented most of the rigidbodies and physics of the game with Matter.js. This took some time, but the entire process was fairly straightforward and intuitive.

Despite this good start, I noticed that there was a major issue: If the ball was moving fast enough, it would occasionally go straight through the player's paddle. I eventually found out that it was because of the way physics engines detect collisions.

Discrete Collision Detection

The general process for the detection of a collision goes as follows: increment physics tick, update object positions based on velocity data, if two objects overlap, make them bounce off of each other. This process is called *discrete collision detection*. It is typically the method used by game engines because of its efficiency. The problem that can occur with discrete collision detection, however, is that if an object is moving too fast, it may move beyond the position of an obstacle between two physics ticks. This is the issue I had when using Matter.js. To resolve this issue, there are typically two options:

1. Make surfaces thicker
2. Increase the frequency of physics ticks
3. Use continuous collision detection (CCD)

Option 1 was not an option for *Push Pong* because the size of players cannot be arbitrarily increased. Option 2 can be a good choice if the maximum speed of any given object is known and low enough that the frequency of the physics engine will not need to be increased too significantly. However, in the case of *Push Pong*, there is no set maximum for the speed of the ball. Hence, I had to go for option 3: continuous collision detection.

Continuous Collision Detection

Continuous collision detection works by looking ahead for possible collisions before changing the position of the object at every tick. This is more computationally expensive, but it ensures that fast objects will not traverse through other objects without causing a collision.

The only problem? Matter.js does not have CCD. In fact, a GitHub issue has been open since 2014, and is still not resolved nearly five years later. Hence, my only solution was to switch to a different physics engine completely. I ended up choosing P2 because I was certain that it would support CCD.

While some of the work transferred fairly easily to the new physics engine, most of the work done with Matter.js had to be thrown out and redone from scratch. Additionally, unlike Matter, P2 does not come with a built-in renderer, so I had to implement the rendering myself. This ended up taking a significant amount of time, and also made debugging much more difficult in the early stages because it was difficult to determine whether problems were due to errors with the physics engine, or errors with my rendering of the physics state.

So the main lesson that I will be taking away from this is that it is generally worthwhile to take the time to look at the most commented on open issues of a library before jumping in and starting to use it. This small amount of due diligence could have saved me tens of hours of development time.

GENERAL NETWORKING

Before getting into the synchronization aspect of the project, I had to set up the general network interactions involved in the creation and state management of a match. In Push Pong's backend, matches are created when a player searches for a match and none are available. When the second player joins, one of the two players is selected as the *host*. The client is never aware of the existence of which player is the host. The benefit of having a host is that the server does not need to run a physics simulation for each game. This results in much lower server costs than other solutions. The downside is that it is possible for players to create movement & positions for the game, although this could be partially mitigated with several anti-cheat measures.

As the game goes on, controls are broadcasted, points are counted, and once the server detects that one player has enough points, it sends an end of game signal to the clients in the match.

Getting matchmaking to work properly went fairly smoothly. Socket.io's namespace feature was used to ensure that messages are broadcast only within a specific channel for each match. The part that was the most difficult was to manage receiving certain messages at unexpected times during a match. To mitigate this, I opted to use a state machine on the server. I used the javascript-state-machine library to accomplish this. That library allowed me to quickly define the different states of a match, and only worry about events by defining valid transitions. The state machine spared me lots of headaches and development time because it was easy to debug invalid transitions. Out of all the decisions I made regarding the architecture of my code, I would say that opting for the usage of a state machine was probably the best one, as it drastically simplified my code, and made it very easy to trace, log and debug. It was such a convenience that I decided to apply the same concept to the client code as well.

SYNCHRONIZATION

Since the game was intended to be played over the internet, steps had to be taken to ensure that both players' game state remained in-sync despite the inevitable communication delays. This concept is generally referred to as *netcode* in the context of game development.

As I briefly described in the beginning of this report, the first resources I looked into for information on netcode architecture mainly discussed snapshots, interpolation, and extrapolation. Hence, I decided to use a combination of the above techniques to synchronize Push Pong client instances. Here is a typical instance of what could be occurring over the course of a game:

1. Client 1 presses *UP* control, updates his own velocity, and sends the control change to the the server
2. Server broadcasts this control to all other players in the match except sender
3. Client 2 receives the control and applies it to the opponent's movement
4. Client 1 releases the *UP* control, updates his own velocity, and sends the control change to the server
5. Server broadcasts the control change once again
6. Client 2 receives the controls, and applies it to the opponent's movement

This interaction works, but obviously, even with really low latency, it will still lead to some discrepancy in the position of client 1 on both of the clients due to the fact that the player's controls are assumed to remain the same until the toggle signal is received (extrapolation). This is a problem because even a small difference in player position could lead to a very big difference in the position of the ball on both clients. Hence, I decided to add *snapshots*. At a set frequency, both clients would send the position of players, as well as the position of the ball. Upon receiving a snapshot from the host, the server would forward the position of the host's player and the position of the ball from the point of view of the host to the other player. Upon receiving a snapshot from the non-host player, only the position of the player would be forwarded to the host. This simple mechanism, if snapshots were sent at a high enough frequency (eg. 10 times per second) allows the game to be playable with an acceptable performance, and appear fairly smooth on the host and client if they both have a fairly low latency with respect to the server.

The problem with this approach is that by the time the client receives a snapshot, it is a snapshot of the past position from the point of view of the other player. This means that even if two clients were perfectly synchronized, the position of the ball on the non-host client would still be slightly reverted back by a distance proportional to the latency of the two players with the server, causing a desync where none was present initially.

This solution tends to work reasonably well for games where the player rigidbodies do not interact with a shared body. For example, first person shooters can achieve a fairly good performance by using similar strategies because the movement of one player, even if inaccurate on one player's environment, will not affect the position of other objects in the world. However, when there are shared rigidbodies such as the ball in Push Pong, more rigorous synchronization is typically needed to ensure better synchronization.

So to summarize, my solution with snapshots and extrapolation was decent, but technically guaranteed that players would always be slightly out of sync by a factor depending on the latency between the two players and the server. Additionally, my choice to select one client as the host is efficient in terms of server cost, but is not optimal for a game that is meant to be played competitively, as it allows for the possibility of movement and position hacks. In general, one should *never* trust the client if they expect the information received to be valid. Hence, there must be a better solution if the goal is to achieve high reliability.

SYNCHRONIZATION IMPROVEMENTS

After implementing this proof of concept, I was still confused about how competitive games managed to avoid the desynchronization issues that my game would suffer even at low latencies. One possible solution would be instead of applying a snapshot directly, to try and approximate where the position should be based on the ping latency and snapshot information received. This technique is known as *lag compensation*. This technique can reduce the visibility of desync issues, but since it is based on *prediction* (extrapolation), it could never fully fix the issue; only mitigate it. One major flaw of lag compensation is that not only could it have the opposite effect and make some situations much worse, it also makes the codebase much more complex because lots of logic has to be set in place to ensure that predictions lead to valid states and rigidbody positions.

So after thinking about this, I returned to the drawing board, and decided to do some more research. I asked some questions on game development forums, and eventually, someone linked me to a talk from the developers of Rocket league at a game developer conference. After rewatching the entire recording several times, I finally got a clear understanding of how competitive games achieve better netcode. I highly recommend watching the video because it provides detailed examples, but the following discussion provides a summary of the idea.

Physics Engine Requirements

First, a deterministic physics engine is needed. This means that given the same inputs, the next state is guaranteed to be the same. This was not guaranteed by the physics engine I used for Push Pong, and from what I can tell, none of the JavaScript-based physics engine provide this feature either. Regardless, this was not the main source of discrepancy, but it is still important to mention.

Second, the physics engine must be able to store a list of states at every tick. At the very least, it should have a buffer of a fixed number of previous states. If this is not implemented by the physics engine itself, it should be coded on top of a current physics engine. For this to be realistically feasible, the physics engine should at least give the ability to snapshot and restore states.

Once the physics engine is set up with these two preconditions, these features can be used to improve synchronization.

Server-side Simulation

On every tick of the client's physics engine, the player controls are sent to the server along with a number identifying the current physics frame. The server buffers the client inputs, and pulls from the buffers whenever it decides to run the next tick. This means that the server's physics simulation is always a little behind the client's, and it does not need to run at a fixed rate. It just moves forward when it has received inputs from both players at the next physics frame.

Technically speaking, the server should aim to only be a little behind the client, so if one client has a very bad connection, it may need to assume its controls remained constant, and only revert back once it receives the controls. *The process for reverting and updating the state is described in the client-side simulation section below.*

Once the server runs the next frame, it compresses the state of the game physics at this frame, and sends that to the clients, along with the number that identifies that frame. This replaces the *snapshots* that Push Pong currently uses.

Client-side Simulation

Meanwhile, on the client side, the simulation continues at a fixed rate while it waits for *snapshots* from the server. Since it does not know what the other players' inputs are until it receives a snapshot from the server, it assumes that the inputs remain constant (or it applies some decay function based on the last input received). The client keeps running the simulation this way until it receives a snapshot from the server. Once it receives a snapshot, it goes back in time to the frame number sent with the snapshot, applies the compressed physics state, and fast forwards back to the current state (while only re-rendering the current state). All of this occurs before the next physics tick, which occurs at a fixed rate on the client-side.

Pros & Cons

The big advantage of this deterministic lockstep is that two players that are already perfectly synchronized will not become out of sync by receiving a snapshot, which was an issue with the method I used for Push Pong. With the labelling of frame numbers and the ability to go back into history and apply changes, it is guaranteed that in-sync clients will remain synchronized, while out of sync clients will be brought back in sync.

The main disadvantage of this method is that this is much more computationally expensive. Not only must the server be able to compute the physics simulation for every running match, clients must do the same and frequently go back in time, apply transformations, and fast-forward back to the current state before the next frame is expected to occur. This all has to occur on top of rendering the game in real-time. This makes it much more computationally expensive for both the server and the clients. Additionally, this method has a much higher load on the bandwidth of both the clients and the server because a request is sent over the network at every tick of each of the client's engines and at every tick of the server's engine.

Despite that, it seems that modern internet connections are capable of handling such heavy network loads, and the physics computations can be manageable as long as there aren't too many rigidbodies simultaneously present in the simulation.

FINAL THOUGHTS

The goal of this research project was to learn about the full-stack of technologies required for the development of a multiplayer game that is playable on a browser. Looking back at all the work I have done, I can confidently say that I have learned a lot from this research project, and it has helped me become a much more well-rounded developer.

WORKS CITED

- Lance.gg architecture of a multiplayer game:
  http://docs.lance.gg/tutorial-overview_architecture.html

- Socket.io documentation:
  https://socket.io/docs/

- Docker 13-Part Tutorial Series:
  https://rominirani.com/docker-tutorial-series-a7e6ff90a023

- Google Cloud Platform guide on mapping custom domains:
  https://cloud.google.com/appengine/docs/standard/python/mapping-custom-domains

- Matter.js open issue for continuous collision detection:
  https://github.com/liabru/matter-js/issues/5

- P2.js documentation:
  http://schteppe.github.io/p2.js/docs/

- Javascript-state-machine documentation:
  https://www.npmjs.com/package/javascript-state-machine

- Jared Cone’s networked physics talk at GDC 2018:
  https://www.youtube.com/watch?v=ueEmiDM94IE

- Code repository for Push Pong:
  https://github.com/DominicRoyStang/pushpong