Nerf Goliath

An adaptive peer-to-peer network protocol for fairer online games.

Mark Pekala, Ream Gebrekidan

May 12, 2023

Abstract

We present an adaptive peer-to-peer networking protocol aimed at fairly balancing latency in multiplayer games. To achieve this we have implemented a game wherein the loser of the game serves as the host server for the game, adaptively switching as users play and the score of the game updates. Overall this implementation succeeds in providing the current loser of the game with the lowest latency and best gaming experience, but more work can be done to assure a quality experience for every player.

0 Code, Engineering Notebook, and Slides

The code for this project can be found at https://github.com/mpekala23/NerfGoliath. All core code exists on the main branch, but a few experiments we're conducted in separate branches so as not to clog up the implementation with unnecessary logging logic.

Our engineering notebook can be found here: Engineering Notebook.

Our slides can be found here: Presentation Link (Note that we did a live demo at the SEAS fair so these slides were merely a small introduction to our live demonstrations)

1 Introduction

Online multiplayer games that require real-time communication present a variety of engineering challenges. One that players care most about is lag. Lag is generally measured as the delay between a player's input and observation of a corresponding state update in the game. Across games, higher lag is associated with a worse player experience, and (although we only observe this informally) a higher chance of losing.

In this project we don't try to solve the problem of lag. Rather, we attempt to balance it's affects by dynamically switching who is coordinating state updates in a peer-to-peer system to give losers an advantage.

1.1 Traditional Multiplayer Networks

Most multiplayer games are supported by a centralized architecture. Every player will send their inputs to a single server where the game is being run and then updates are sent back to the user. This has a number of benefits. Firstly, the game developers can ensure that the actual machine performing the state updates and running the game is high quality and can be replicated if needed. Secondly, a single server can more easily and efficiently coordinate communication with a large group of players. This inherently makes centralized architectures more scalable than peer-to-peer systems.

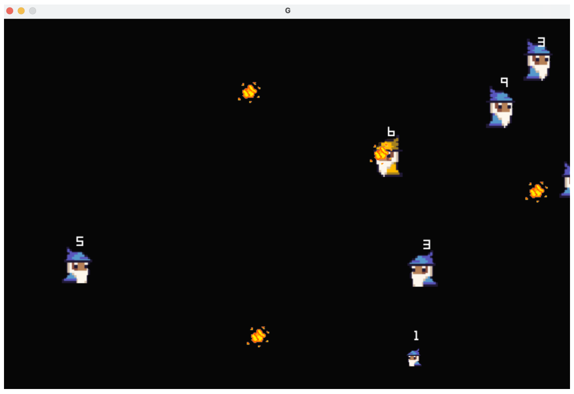
Figure 1: A screenshot from "Nerf Goliath".

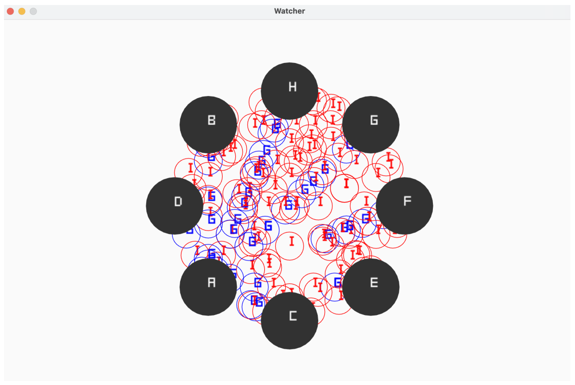

Figure 2: A simple visualization of network traffic in "Nerf Goliath". Blue circles represent game state updates and red circles represent input updates.

From an individual player's experience, however, a centralized architecture can have it's drawbacks. On low or even moderate quality networks, sometimes sending packets to and from the central server simply take too long to feel good, even if all the players are geographically close. This is made worse by the fact that many game studios tend to only put servers on the coasts, leaving out a large number of players in the middle of the country.

For smaller studios, the cost of centralized game servers both financially and in development time can be overwhelming[Tra21]. In some of these cases, a peer-to-peer network in which players communicate state updates directly with each other may be an attractive option.

2 Implementation

2.1 Goals

We have three main goals for our peer-to-peer protcol:

1. Great gameplay for the player in last. This is the most imporant goal by far. At any point, whoever's in last should experience no lag and have a minor competitive advantage over the other players.

2. Minimally invavise transitions. This goes hand-in-hand with goal 1. If you are the player who just entered last place, you should notice a better experience almost immediately. If you are a player who is not in last place when a switch occurs, you shouldn't notice that a switch happened. In other words, the additional delay caused by the switch should be roughly the same order of magnitude as the normal delay that exists in the system, and it should exist for a short period of time.

3. Reasonable quality for everyone. Finally, the ideal game should feel good to play for everyone. The advantage gained by the player in last should be a slight nudge, not a game-breaking gimmick.

2.2 Game Design

Designing a game worth playing was the hardest part of this project. To highlight the aspects of our network protocol, we wanted to pick a game that required quick inputs, with a clear notion of score that changed frequently but not *too* frequently.

We decided to model our game after a game we enjoyed called "Boomerang Fu" which involves fruits running around trying to cut each other with flying boomerangs. But instead of fruit we chose wizards, and instead of boomerangs we chose spells.

Players score points by hitting each other with spells. Whoever is in last place has a smaller character and moves slightly faster and thus is harder to hit. This clearly visually communicates to the other players who the current losing player is, and also gives them a slight competitive advantage to increase the odds they'll crawl their way out of last.

2.3 System Design

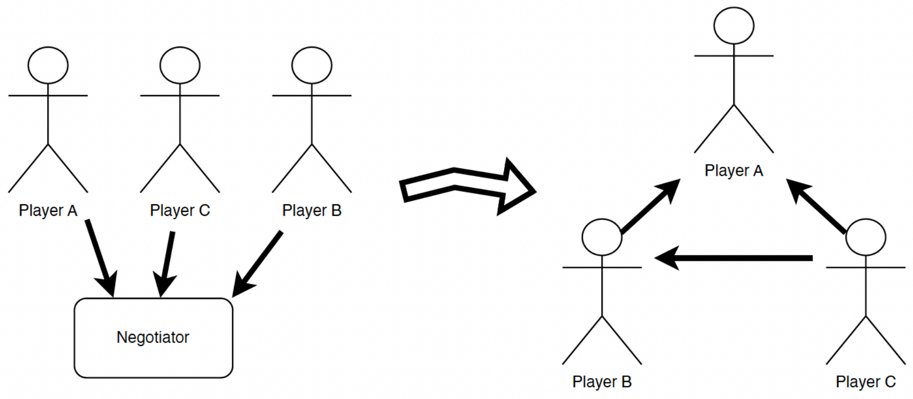

Figure 3: A high level over view of the system design.

During gameplay, our system will simply involve a fully connected graph of all the players. However, to be able to set up such a system dynamically, we needed some kind of service which could allow connections from multiple people and coordinate their names and addresses so communication can be established. This is achieved in our "negotiator" service. Then, we'll need a way of quickly and reliably changing hosts in response to a change in score.

2.4 Negotiator

The negotiator runs on a constant IP and port that is known to all players and is constantly looking for connections (until the max number of players is achieved). The listening happens on a single thread so that the connections from different players are guaranteed to happen in sequence.

When the negotiator receives a request to connect from a new player, it receives both their address and what name they would like to use in the game. If the name they would like to use is already taken, an error is returned to the client, and they can try again with a different name. If their name is unique, then a success message is returned, and the client begins waiting (in a blocking manner) until they receive a start message from the negotiator.

One benefit of having the negotiator sequence all connection requests is that it provides us a very simple way to pick the first leader. The negotiator picks the first leader for the game as simply the first person who connected.

Once the appropriate number of players have connected to the negotiator, the negotiator calculates a series of connections that would establish complete communication between the players. In the start message that the negotiator sends to each client, it also includes a list of IP/port combinations that that player is responsible for connecting to to ensure complete communication between players.

For example, in the right hand side of Figure 3 we see an example of what this communication establishing procedure might look like for three players. Player B has been told by the negotiator that they are responsible for connecting to Player A (and given A's address), while Player C has been told that they are responsible for connecting to Players A and B (and given address information). Player A is not given anyone they must connect with, and instead just listens. This coordination by the negotiator gives us a very simple way for an arbitrary number of players to establish complete communication in as few messages as possible.

Along with this start message, the negotiator also sends out the name of the first leader, so that all players are on the same page at the beginning of the game.

2.5 Host Switching

At the heart of our game is the protocol for how the host switches between players in response to game state. We started with a rather naive approach that worked locally, but had serious flaws and ran into efficiency problems when being run over a real network. Then, we implemented two improvements on top of our original naive strategy which fixed the bugs and improved the efficiency of the protocol.

2.5.1 Naive Switch

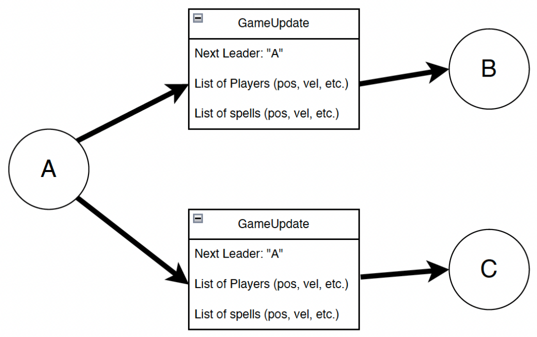
GameUpdate
Next Leader: "A"
List of Players (pos, vel, etc.)
List of spells (pos, vel, etc.)
GameUpdate
Next Leader: "A"
List of Players (pos, vel, etc.)
List of spells (pos, vel, etc.)
A
B
C

Figure 4: Normal operation in the naive scheme. A is the leader and their updates include that A will be the next leader as well.

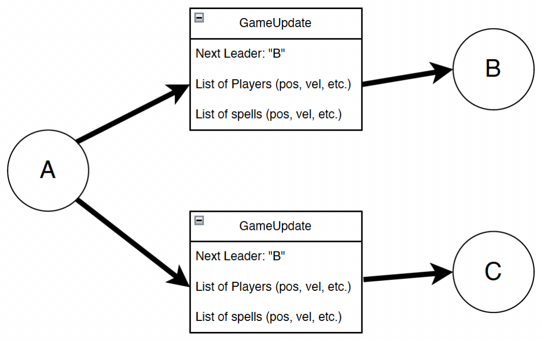
GameUpdate
Next Leader: "B"
List of Players (pos, vel, etc.)
List of spells (pos, vel, etc.)
GameUpdate
Next Leader: "B"
List of Players (pos, vel, etc.)
List of spells (pos, vel, etc.)
A
B
C

Figure 5: The switch update in the naive scheme. A is still the leader, and during a single update to all players informs everyone that B will be the next leader. After this, A immediately stops sending updates.

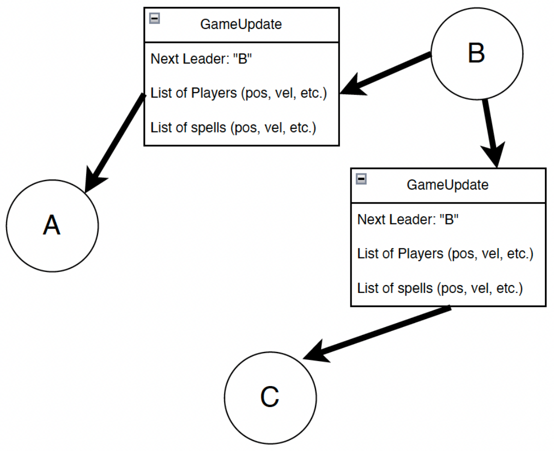
GameUpdate
Next Leader: "B"
List of Players (pos, vel, etc.)
List of spells (pos, vel, etc.)
B
GameUpdate
Next Leader: "B"
List of Players (pos, vel, etc.)
List of spells (pos, vel, etc.)
A
C

Figure 6: B, after receiving the update from A recognizes that it is the next leader. B updates it's state to match the last state received from A, and begins performing state updates and broadcasting the results as the leader.

To get a preliminary version of host switching to work, we started by assuming that communication is reliable (later strategies weaken this assumption).

Under this assumption, we developed a relatively straightforward switching protocol based on the notion of keeping "next leader" information attached to every GameState update. The procedure is shown in detail in Figures 4 - 6.

Essentially, as the name suggests, the "next leader" information on each update tells all listening players who they should listen to for the *next* state update. For the majority of state updates, the "next leader" is the same as the player currently sending the state update. This informs following machines to continue following the same machine, and that no change is needed.

However, once the current leader determines that they are no longer in last place, on their next state update they set "next leader" to the new leader. **In the naive approach, after the old leader sends this update, they immediately stop sending game updates and begin acting as a follower, waiting for a message from the new leader.** In the example above, nothing happens in the system until B receives A's message informing B that they are the next leader. After this, B updates it's state to match the last state sent by A and begins broadcasting messages as the host.

For followers, each player has a variable that stores their current view of who the leader is. If they ever receive a game update from a player that does not match their current view of the leader, they ignore it. Every update from the current leader that matches their view, after updating their state they set their view of the current leader to match the value of next leader included in the update from the old leader.

2.5.2 Problems with the Naive Approach

This simple approach worked locally and was a good place to start, but it has a few serious issues.

The first issue is that depending on the speed that messages get delivered, the system can end up in deadlock. To see why, consider a game with three players, A, B, and C, where A originally starts as the leader.

Say that A realizes that B should be the new leader and broadcasts a message to B and C containing "next leader = B". Next, assume that B receives this information before C, and B starts acting as the leader immediately. One or two frames happen, and — uh oh! — B realizes that C should now be the new leader. So B sends an update to A and C with "next leader = C".

Now, we assume that A's message to C still hasn't been delivered. It's entirely possible that B's update to C telling it that C should be the new leader arrives *before* A's update to C telling it that B should be the new leader. In this case, C would ignore the message from B, and when it did finally receive the message from A, it would end up thinking B is the leader. However, A and B would end up thinking C is the leader, and no more progress would happen in the system.

This is obviously a huge problem, and required us to develop more sophisticated protocols outlined below. The above problem also appears if either of the leader change updates fail to send without proper retry logic.

Another smaller issue with the naive strategy is that it makes the transition more jarring than necessary. There is a prescribed period of time where nothing is happening, which may introduce easily noticeable lag. Ideally we would like to have a strategy where there is less expected delay between the last update sent by the old leader and the first update sent by the new one.

2.5.3 Solution One: Limit the Rate of Updates

For this solution, we assume that there is some fixed amount of time that is the longest possible time that it could take for a message to get delivered. Say this time is equivalent to the time it takes for a machine to process N frames of the game. By simply making it so that new leaders are required to wait at least N frames before picking a new leader, we solve the deadlock condition above.

To see why this solves the problem, say that A broadcasts that B is the new leader at frame 0. By frame N, C is then guaranteed to know this information. The earliest frame that B could possible broadcast a new leader is N + 1, and so by the time B might make a leader change, everyone will know that B is the leader and this state update containing the new leader will not be ignored.

2.5.4 Solution Two: Attach a Logical Clock to Leader Changes

Rate-limiting changes is a quick and dirty solution that works in theory, but sets us up for trouble in practice if our chosen N is not high enough. A better solution involves making leader change updates dependent on a logical value, and making it so that followers compare the tick on that logical value when deciding whether to accept a leader change instead of comparing it to their current leader [Lam78].

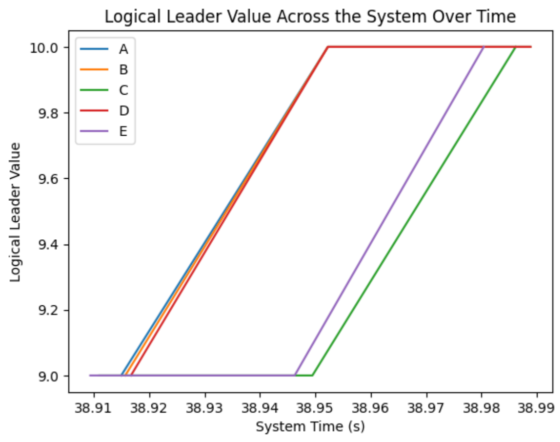

Figure 7: The amount of time take for all machines to get the logical update in a system with no introduced lag.

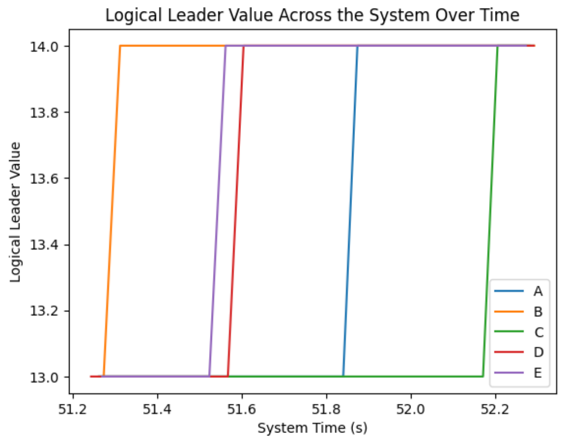

Figure 8: The amount of real-time taken for all machines to get the logical update in a system with 100ms+ of lag.

Figures 7 and 8 show how long these logical updates take to be stable across players in various lag settings. The introduction of 100ms of simulated lag slows down the amount of time it takes for the last player to learn of the update by around 8x.

2.5.5 Fixing a Retry Bug and Improving Performance

With the above implemented, we observed one last bug in our system that only very rarely appeared when testing over a real network with large delay.

Sometimes, the message from the old host to the other players about the new host would fail to be delivered at all, and our system had no way of rectifying this. We realized that this was because of a decision we had made earlier to improve leader performance.

In our original leader logic, if a game update failed to send, we would simply ignore the failure and move on to the next frame. Logically, this makes sense, as frames happen 30 times a second, so wasting time and energy retrying a failed update doesn't necessarily make sense as that information will be immediately invalidated by the next frame.

However, if that game update contains a leader change and it fails to send, moving on to the next frame without doing anything else spells disaster as now no player will think they are the leader.

At the same time we also noticed a performance issue in our system where during switches some players would see a very big “jolt” on the screen as their position was snapped to the new hosts state.

We solved both these issues at the same time by implementing one last piece of logic in our leader code surrouding switches. Before, as soon as a player stopped being the leader, they would stop sending out game updates. Now, once a player stops being the leader, **they continue to send out game updates until they receive their first game update from the new leader**.

This solves the send failure issue above as if one of the old leader's messages to the new leader ever fails or gets dropped, it will keep sending it until the new leader both receives it and starts sending out game updates.

To see why this improves performance, say that A, B, and C are playing the game and A is the first leader. In the original system, once A stopped being the leader, C wouldn't receive it's next game update until a message went from A to B, and then B to C. In the improved system, since there is a brief period of time where both A and B are sending game updates, C may get additional game updates from A in the time it normally would have been waiting for B. While it is true that these extra updates from A may not be exactly what they later receive from B, A is still processing input and so it's likely that any additional updates it receives will be better approximations of the state it will eventually receive from A, and thus lead to less of a "jolt" when the final state from B is received.

One might wonder if the fact that there exists time when both A and B are sending game updates might lead to thrasing between the two states. Since we implemented a logical leader counter above, this is not a problem. A player will believe that A is the leader until it receives it's first message from B. At that point, the value of the logical leader counter will increase, and any future messages from A will be ignored, so the player will not thrash between A and B's view of the game state.

2.5.6 Tying it All Together

Our final host-switching logic was a mix of all of the following. We have a minimum time (one second) that must pass between leader changes. Each leader change increments the logical leader counter and updates from leaders with a lower count are simply ignored. We also make it so that old leaders continue to act as the leader until they hear from the new leader, improving both robustness and performance.

3 Results

3.1 Input Efficiency

Once the game was complete and working, we turned our attention to improving efficiency and benchmarking performance.

This section describes a simple but important optimization we made over our initial design which helped the game scale to more players. Originally, we broadcasted player input on regular inputs, roughly at the frequency of game ticks. To simplify the logic of host switching, we had input broadcasts occur between all players all the time, so that as soon as a new leader takes control, it already has an up to date input map and doesn't have to wait to receive inputs from every player.

This worked great for 2-4 players, but beyond that the game performed noticeably worse. To improve this, we instead broadcasted inputs only on change, which is likely substantially less than the number of frames.

To test this, we spun up a game with 2-6 players (all of which were controlled by simple AI choosing relatively random input) and measured how many input communications occurred using both schemes. Unsurprisingly, we observed a substantial reduction in the amount of communication occurring and generally a smoother experience.

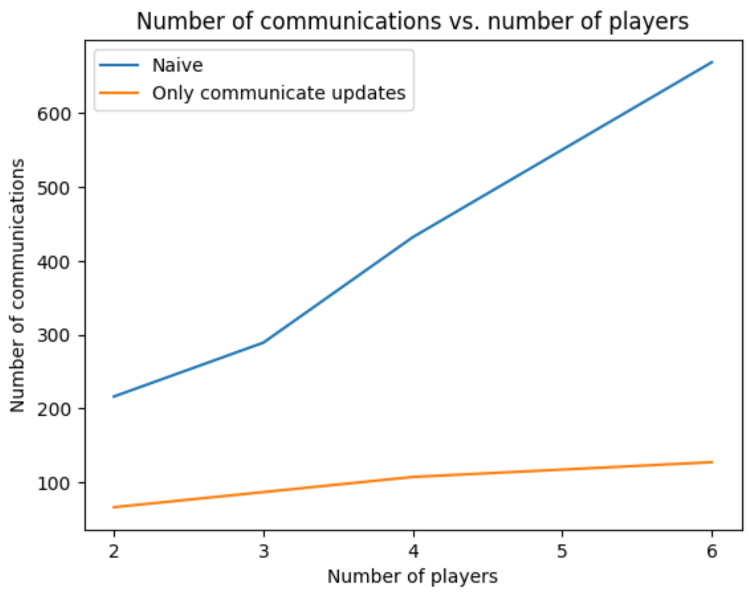

Figure 9: A graph showing the number of communications in the system for the naive strategy (sending updates alongside gamestate) vs. only sending updates on input change.

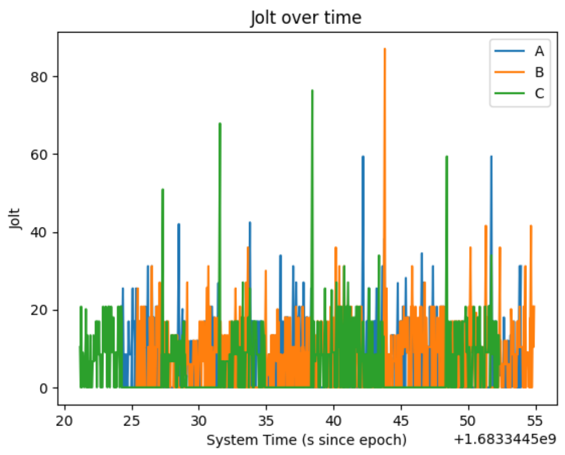

Figure 10: A graphing showing the jolts received by the system during a game played between three people.

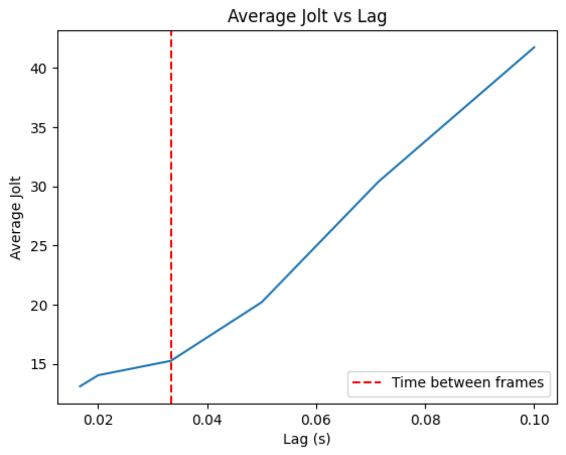

Figure 11: A graph showing how the average jolt varies by lag.

3.2 Jolt

When benchmarking our game, one of the things we cared most about was answering the question: “how obvious are leader changes?” We care about the answer to this question from the player’s perspective, which boils down to instead asking: “how much will my character ‘jump’ or ‘jolt’ to an unexpected location during a leader change?”

This led us to develop the notion of *jolt* as a way to quantify the observability of leadership changes. Formally, jolt is defined as the sum of the distances between between each players predicted location and their actual location as received over the wire from the leader.

In Figure 10 we look at the jolt of each update during a game played between three people. On the x-axis is system time, and on the y-axis is the jolt received. The portions of the graph that are straight horizontal lines at y=0 represent areas where no jolt is logged (because that player is hosting the game).

Interestingly, we observe that significantly high jolts do occur, but they occur for the player who just *stopped* hosting. An example of this occurs in the graph around a system time of 38 seconds. There is a large green spike immediately after a period of time when green was hosting. We are not entirely sure why this spike occurs, and why it is so much more sever than what the third player in the system observes. It potentially could be due to the fact that in our third optimization for switching we had the former host continue to update game state for a period of time after relinquishing control. This may have created more situations in which the old hosts state will derail from the new hosts state substantially more than machines that just follow because the old host will have access to their own inputs without delay, whereas the new host does not.

In figure 11 we study how the average jolt in the system changes as we introduce more and more lag. Unsurprisingly, jolt increases relatively quickly as the lag increases. One interesting result, however, was that the rate at which the jolt increased substantially increased once the amount of lag exceeded the amount of time between update frames. This makes sense, as when the lag is below that number, it’s likely that many of the inputs, even with added lag, would arrive *before* the next state update, reducing the discrepency observed by the follower.

Having jolt be zero when the player is the host shows that we have successfully accomplished the first goal of our system, giving the last place player a great experience. Having jolt be relatively constant for players not participating in leader switches (i.e. neither new / former host) means that we’ve achieved the second goal of the system, which focused on having minimally invasive transitions. Accomplishing the third goal is still somewhat up in the air, and is discussed more in the challenges section.

3.3 Lag Measurement

To achieve our first goal of giving the player in last place the best gameplay experience they should, in theory, have less lag. That is, their inputs should take less time to affect the actual game state compared to any other player. To measure this we recorded the times at which a user inputted a control, in this case the up movement key, and then measured the time at which the game state was altered with the inputted movement. As seen in Figure 12, on average the leader will have 0.01614 seconds less lag compared to a follower at any given time. This measurement was an average across multiple inputs/runs with the leader/follower switching halfway through the testing.

4 Challenges and Next Steps

4.1 Achieving Good Quality for Everyone All the Time is Hard

Our third goal was to have a good game with minimal lag for everyone all the time. This was hard. As was shown in Figure 10, the average jolt for non-leader players is around 15-20, but spikes up somewhat regularly. This corresponds to shifting of the players position by roughly 1/4 of their width every few seconds, which is fairly noticeable.

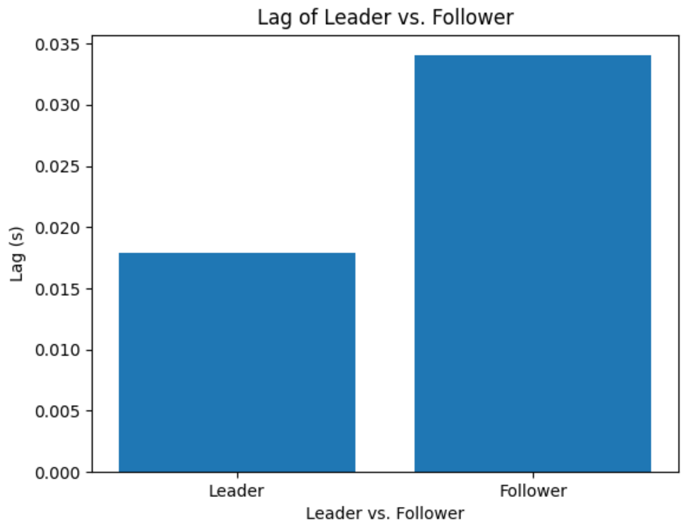

Figure 12: A graph showing the time between when a user inputs a control and the game state is updated.

In real games, there is often complex logic dedicated to predicting the location of players and objects between updates from the leader. This is known as *dead reckoning*. We implemented a basic form of dead reckoning which involved predicted straight paths for everything in the system, but did do anything more complicated like predicting hits or boundary collisions. Perhaps with more complex dead reckoning the experience would be better.

4.2 More Efficient Wire Protocols

Because of the exploratory nature of this project, once we had a working wire protocol, we didn't spend time optimzing it. We are certainly sending far more bytes than necessary on every update. In the future we also could explore sending packets using UDP instead of TCP, although this would require more logic to rectify packets as UDP provides fewer guarantees about packet delivery and ordering.

References

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Tra21] Philip Trahan. Among us server issues resolved, 2021.